OpenUIDL, a User Interface Description Language for Runtime Omni-Channel User Interfaces

ALEX MOLDOVAN, VLAD NICULA, IONUT PASCA, MIHAI POPA,
JAYA KRISHNA NAMBURU, ANAMARIA OROS, AND PAUL BRIE, teleportHQ, Romania

We extend the concept of cross-device user interfaces into the new, more general, concept of omni-channel user interfaces to better reflect the technological variety offered for developing multi-target user interfaces for interactive applications. We present a model-based approach for developing runtime omni-channel user interfaces for multi-target applications, which consists of: (1) OpenUIDL, a user interface description language for describing omni-channel user interfaces with its semantics by a meta-model and its syntax based on JSON, (2) the definition of a step-wise approach for producing runtime interactive applications based on OpenUIDL with integration into the development life cycle, (3) the development of a cloud-based, OpenUIDL compliant, Interactive Development Environment that supports the application and the enactment of the step-wise approach and its illustration on several multi-target user interfaces.

CCS Concepts: • **Human-centered computing** → **Graphical user interfaces**; *Interactive systems and tools*; • **Software and its engineering** → **Domain specific languages**; **Graphical user interface languages**; **System modeling languages**; *Model-driven software engineering*; *Source code generation*; *System description languages*; Interpreters; Runtime environments; Extensible Markup Language (XML); Interface definition languages; • **Computing methodologies** → *Model development and analysis*;

Additional Key Words and Phrases: Model-based user interface; Multi-target user interfaces; Omni-Channel user interfaces; Open source; User interface description Language

ACM Reference Format:
Alex Moldovan, Vlad Nicula, Ionut Pasca, Mihai Popa,, Jaya Krishna Namburu, Anamaria Oros, and Paul Brie. 2020. OpenUIDL, a User Interface Description Language for Runtime Omni-Channel User Interfaces. *Proc. ACM Hum.-Comput. Interact.* 4, EICS, Article 86 (May 2020), 52 pages. https://doi.org/10.1145/3397874

1 INTRODUCTION

In the field of Engineering of Interactive Computing Systems (EICS), the term "User Interface Markup Language" (UIML)[1] has been initially used to refer to a marker language for user interfaces [63], later being replaced by the term "User Interface Description Language" (UIDL) to refer to a meta-language for describing, specifying, and analyzing a user interface independently of any implementation [104]. Many markup languages (*e.g.*, HTML, XML) and programming languages (*e.g.*, Java, C#) co-exist to implementing the Graphical User Interface (GUI) of an interactive system and they all have their own peculiarities. To raise the level of abstraction and the independence with respect to these diverse, heterogeneous, languages, several UIDLs have been defined and used [43, 66, 107].

[1]Not to be confused with the eponym language: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml

Author's address: Alex Moldovan, Vlad Nicula, Ionut Pasca, Mihai Popa,
Jaya Krishna Namburu, Anamaria Oros, and Paul Brie, teleportHQ, Calea Moților, 84, Cluj-Napoca, 400370, Romania, {alex.moldovan,vlad.nicula,ionut.pasca,mihai.popa,jaya.namburu,anamaria.oros,paul.brie}@teleporthq.io.

2573-0142/2020/5-ART86
https://doi.org/10.1145/3397874

Standardization and dissemination efforts have been devoted for some of them: XIML [34] by the XIML Consortium, UIML [44] by the OASIS Specification Committee, IFML [17] by OMG. Several UIDLs, such as MariaXML [93], UseML [66], and UsiXML [62], joined their effort under the umbrella of the W3C Group on Model-Based User Interfaces [65, 96]. Despite these fruitful efforts, apart some exceptions, most UIDLs have known little or no usage to feed commercially available software. A renewed interest for UIDLs is born with the multiplication of languages to implement UIs, their successive versions, the platforms, devices, and their operating systems which perpetually evolve over time. Merely for developing UIs for the web, developers face a wide variety of software architectures, languages, packages, and libraries that are today available, among which choosing the right one is challenging.

Several potential shortcomings may explain why the impact of such UIDLs remains under its expectations for today's highly interactive applications:

- S_1. A *limited accessibility of their definition*: UIML and IFML are probably the most documented UIDLs as they are promoted by their respective standardization bodies, *i.e.*, OASIS and OMG. The W3C specification for Model-based UI is equally documented, but requires a thorough UIDL knowledge to effectively transfer these recommendations into a practical UIDL for commercial usage. This working group has however defined a precise vocabulary [96] in the domain. XIML [35] is a proprietary UIDL requiring a consortium affiliation and paying some fee or royalties for any commercial usage.
- S_2. A *loose integration into code generators*: confronted to the constant evolution of development practices, the code generators implemented to transform a UI description into code do not follow the same pace and are often a war behind, thus limiting their attractiveness. Moreover, editing a UI is often performed in an editor that is external to the generation engine, thus preventing a tight coupling between editing and generation.
- S_3. A *narrow generation bandwidth*: most UIDLs offer code generation for a few target platforms (*e.g.*, HTML UIs for a desktop and a smartphone), with little or no variation or power for any such platform (*e.g.*, HTML and CSS2 only for browser UIs), without taking benefit from multiple advances such as jQuery [111], Google's Material Design [71], React.js for UI [14]. This shortcoming is often referred to as the problem of "high threshold, low ceiling" [82] with "narrow walls" [112] (fig. 5, p. 7).
- S_4. A *limited language expressiveness*: raising the level of abstraction is subject to the risk of loosing expressiveness and, thus, control over the UI options. For example, some UIDLs offer a limited amount of UI patterns [77] or UI design options [113], resulting from clearly defined conceptual constructs. But this is not a common practice. Lack of appropriate conceptual modelling could result into an over-simplification of UIDL expressiveness or its understatement.
- S_5. A *limited support for runtime user interface*. Most UIDLs and their Integrated Development Environments (IDEs) generate user interface from specifications written at design time. Consequently, they offer limited support for producing similar user interfaces at linking, compile, and runtime. User interfaces requiring some form of adaptation at runtime are therefore outside the scope of these environments. This is why the Models@Runtime initiative [11] has fostered the runtime usage of model for generating code at runtime.
- S_6. A *limited integration into the rest of the interactive application*: many UIDLs can automatically generate UI code, but with little or no integration with the application architecture. Consequently, the developer is responsible for ensuring a tight coupling, which is very demanding in terms of software engineering knowledge and experience.
- S_7. A *lack of efficient software support*: only a few UIDLs are delivered with their own Integrated Development Environment (IDEs), which should include an efficient UIDL editor, a rendering

engine, and efficient UI interpreters and code generators. These software are particularly complex to develop and to maintain with respect to the evolution of the technology.

S_8. *No support for omni-channel user interfaces*: UIDLs have progressively evolved from a single target (*e.g.*, generating UI code for one language for one platform) to multiple targets (*e.g.*, generating different UI codes for different platforms and languages). But, again, their "walls are narrow". To the best of our knowledge, none of them support the description and the generation of user interfaces according to the omni-channel paradigm, that emerges today beyond the paradigm of cross-channel or distributed-channel paradigms.

To address these shortcomings S_1 to S_2, this paper provides a model-based approach for developing runtime omni-channel user interfaces based on a new UIDL called OpenUIDL. A model-based approach for developing an interactive application is typically composed of three pillars [66]:

(1) *Models*: one or several models represent the interactive system at a higher level of abstraction than programming code. They models cover various aspects such as the end user, the computing platform, the organizational environment, the tasks, the data, and the processes. To preserve rigorousness, each model should be compliant with a *meta-model*, that regulates how models of the reality could be expressed. This meta-model is represented through a language to express any model, each language having its own notation.
(2) *Step-wise approach*: the aforementioned models are created, edited, and exploited according to a structured approach, which is decomposed into steps, to reach the final application.
(3) *Software support*: since applying the approach manually requires some experience, software should support the application of the approach to a particular interactive application.

To cover these three aforementioned pillars, the contributions of this paper are as follows:

(1) A motivation and definition for the new concept of omni-channel user interfaces (S_8).
(2) The definition of OpenUIDL, a UIDL for describing runtime (S_5) omni-channel user interfaces (S_8), based on its semantics and its syntax. This OpenUIDL is open, accessible (S_1), and expressive enough to produce such UIs (S_4).
(3) The definition of a step-wise approach for producing runtime interactive applications based on OpenUIDL, which provides a real integration into the development life cycle (S_6), particularly the code generators (S_2)
(4) The development of a cloud-based, OpenUIDL compliant, Interactive Development Environment that supports the application and the enactment of the step-wise approach (S_3, S_7, S_8) and its illustration on several examples, including video demonstrations.

The remainder of this paper is organized as follows: Section 2 revisits the notion of the UIDL in the light of recent advances in user interface development, discusses its various aspects, and define the concept of omni-channel UIs. Section 3 defines the meta-model of the OpenUIDL in terms of a UML Class Diagram and explains its major modelling constructs. It also defines the OpenUIDL notation in terms of a JSON format for expressing an omni-channel user interface. Section 4 describes the step-wise approach in terms of SPEM notation [91] and the software environment for developing interactive applications having such UIs. Section 5 revisits the eight initial shortcomings in the light of the work presented to compare potential benefits with respect to observed shortcomings. Section 6 concludes this paper by highlighting the significant progress made with OpenUIDL and its current limitations to be addressed in the near future.

2 RELATED WORK AND BACKGROUND

2.1 Overview of Multi-Target User Interfaces

The creation of a Graphical User Interface (GUI) takes generally 48% of the source code, 45% of the development time, 50% of the implementation time, and 37% of the maintenance time [83]. Consequently, 44% = $\frac{45+50+37}{3}$ of the total time to create an interactive system is estimated for developing GUI [83]. Although this figure was obtained in 1992, it is still topical. Results from the Standish Group's research[2], show already for many years that interactive software development project fails or underperforms in almost two thirds of the cases and that the main reasons for these failures can be attributed to requirements engineering problems such as the lack of user involvement, incomplete user requirements, and change due to platform evolution. The constant evolution of GUIs, the new interaction modalities such as gesture, 3D, vocal, Brain-Computer-Interaction and the need for interoperability between these modalities wold probably maintain the figure at the same level.

When it comes to deal with the vast set of physical platforms, like mobiles phones, smartphones, tablet, laptop, desktop, GUIs should be implemented considering a wide spectrum of programming and markup languages differing in syntax and abstraction such as XML, JavaScript, Java C++. Each of them comes with its own particular syntax. For example, the HTML 4.0 syntax for creating a push button is "<button>" while the Swing library for Java specifies JButton b = new JButton;.

When it comes to adapting GUIs [51] to the different types of end users, all with their with preferences, abilities, cultures and languages, the design space explodes in terms of possibilities to be considered. This does not mean that all the alternatives should be designed and compared, but that a careful selection should take place.

When end users are mobile, moving around or transitioning from one location to another, developers have also to deal with different working environments, each environment imposing its own set of constraints ranging from environmental conditions to organisational setups. For example, the ambient noise or the luminosity in an office room in addition to locations or various formats used for times.

These elements are grouped together under the umbrella *context of use*, which is defined as a set of (end user, platform, environment) [19]. Nowadays, the variety of context of use is really large, is dynamic and has consequently to be considered [66], thus giving rise to the concept of multi-target user interfaces [19, 23]: multiple user interfaces, along with their variations, need to be developed to cover all expected variations of the context of use. A *multi-target user interface* is hereby referred to as a single UI or a set of related UI that address the constraints imposed by several contexts of use, which cover variations in terms of platforms, users, and environments. It is called multi-target in the sense that it pursues multiple targets at once. Consequently, developing multi-target UIs is even more time- and resource-consuming with more and more challenging schedules than a single-target user interface.

Developing manually a multi-target user interface is challenging as long as it subsumes developing multiple UI instance manually and separately of each other. In particular, this approach requires some coordinated development organisation in order not to duplicate efforts on common parts, while acknowledging the specific aspects of these instances.

Instead of relying on a manual, uncoordinated approach, the paradigm of *Model-Based Design User Interfaces* (MBUI) [66] wishes that such UIs could be expressed in a UIDL, a sort of meta-language that would not vary over time to remain as autonomous as possible from technological evolution and that would be able to express contextual variations. Different benefits of MBUI design can be observed for the stakeholders (see [65] for more details).

[2]See https://www.standishgroup.com/chaosReport/index for its last edition.

2.2 Definitions of UIDL over Time

No universally accepted definition of what a UIDL actually is exists today. So, before giving our own working definition, we compare various definitions extracted from several historical sources in order to give different viewpoints on UIDL (See Table 1). We observe in this table that UIDL definitions share a certain amount of reflections. A UIDL is basically a language for specifying a user interface, but that can be used in any stage of the development, not just for coding, especially for facing the problem of multi-target user interfaces, which is probably more acute for user interfaces than for the rest of the application.

Date and source	Definition
Souchon & Vanderdonckt (2003, p. 377) [107]	"A UI Description Language (UIDL) consists of a high-level computer language for describing characteristics of interest of a UI with respect to the rest of an interactive application. Such a language involves defining a syntax (*i.e.*, how these characteristics can be expressed in terms of the language) and semantics (*i.e.*, what do these characteristics mean in the real world). It can be considered as a common way to specify a UI independently of any target language (*e.g.*, programming or markup) that would serve to implement this UI."
Luyten *et al.* (2004, p. 2) [63]	"A UIDL must be a canonical meta-language that supports model-based user interface development [...] by separating concerns, and which must identify 'presentation logic' as first class entity, and represent it in a portable device-independent manner to allow compilation to target language."
Shaer *et al.* (2008, p. 4) [104]	"A user interface management system (UIMS) allows designers to specify interactive behavior in a high-level user interface description language (UIDL) that abstracts the details of input and output devices. This specification would be automatically translated into an executable program or interpreted at run time to generate a standard implementation of the user interface. The choice of a UIDL model and methods is a key ingredient in the design and implementation of a UIMS."
Guerrero *et al.* (2009, p. 36) [43]	Same definition as in [107] plus "in order to be used during some stages of the UI development life cycle" + "A user interface description language (UIDL) consists of a specification language that describes various aspects of a user interface under development."
Fonseca *et al.* (2010) [38]	"A UIDL is a formal language used in HCI in order to describe a particular UI independently of an implementation technology. As such, the UI might involve different interaction modalities, interaction techniques or interaction styles. A common fundamental assumption of most UIDLs is that UIs are modelled as algebraic or model-theoretic structures that include a collection of sets of interaction objects together with behaviours over those sets. A UIDL can be used during requirements analysis, systems analysis, system design, runtime."
Guerrero *et al.* (2011) [42]	Same as definition [107] + "in order to be used during some stages of the UI development life cycle."

Table 1. Definitions of an UIDL.

Inspired by these definitions, we hereby refer to a *User Interface Description Language* (UIDL) as any formal language used throughout the development life cycle of an interactive application to describe its user interface at a higher level of abstraction than any programming or markup language used for developing it. The UIDL is aimed at performing any reasoning analysis on any description with the ultimate goal of producing the final running UI for the interactive application, either by interpretation or by code generation. While this goal is recognized to be the ultimate one, a UIDL is not restricted to this usage exclusively. It can be used for forward, reverse, or lateral engineering, as well as for model analysis and checking of properties.

The following non-exhaustive alphabetical list[3] of UIDLs found in the literature or in the commercial domain demonstrates the interest for this question: AAIML [119], AndroidXML [79], ComposiXML [60, 61], DISL [103], GIML [54, 55], HCIDL [40], IMML [57, 58], InTml [36, 37], ISML [24], Transformations [3, 4], MariaXML [93], MDML [48], PlasticML [100], QUID [75, 76], QML [99], QuiXML [113], RIML [41, 118], SeescoaXML [64], SISL [6], SMUIML [31, 32], SunML [28, 89], TeresaXML [10], UIML [1, 2], UIPLML [87], UseML [65, 67–69, 120], UsiXML [62], WSXL [20], XAML [84], XCML [86], XICL [25, 26], XIML [34, 35], XISL [53], XMMVR [90], XooML [49, 50], and XUL [97]. As this list is impressive, conducting a Systematic Literature Review (SLR) of all these UIDLs, even a simple comparative analysis, is really challenging and beyond the scope of this paper. Instead, we refer to the literature for discussing specific aspects. A review of most UIDLs, but not all, has been initiated since 2003 [107] and subsequently updated in 2009 [43] and 2011 [42], but mainly their expressiveness in terms of semantic concepts structured according to the Cameleon Reference Framework (CRF) [19] was analyzed, and not their efficiency to produce final UIs. Other reviews and surveys compare UIDLs on some particular dimensions.

For example, Jovanovic *et al.* [51] reviewed UIDLs in the light of the transformation language used for supporting model-driven engineering of multi-platform UIs. Mayer *et al.* [21] compared UIDLs against three characteristics to support UI adaptability: accessibility to produce UI adapted to user's abilities or disabilities, platform used for executing the UI, and context awareness. Mitrovic *et al.* [73] compared some UIDLs, including commercial ones like XAML [84], AndroidXML [79], and QML [99] against their capabilities for producing runtime UIs on mobile devices: support for multiple platforms and operating systems, multi-language, visual editor, friendly markup language, and layout support. Ruiz *et al.* [101] performed a SLR on an outlet of 96 papers to compare works in MB-UID, some of them exploiting a UIDL to identify various operations involved in MB-UID and Model-Driven Engineering (MDE) such as: forward engineering or reification, reverse engineering or abstraction [15], re-engineering [16], lateral engineering [80], model mapping [78], model transformation [3, 4], plasticity [100] composition [18, 60, 61], distribution [70], guideline review [9], sketching [102], multimodality [53, 59], context-aware adaptation [80], metamodelling [89], accessibility [39], and GUI layout generation [98]. Some workshops [63, 104] also captured some snapshot of UIDL progress in research and development.

2.3 Evolution of Multi-target User Interfaces

Figure 1 graphically depicts when UIDL appeared on a time line. Cousin [109] is the first UIDL that appeared in HCI history, mainly dealing with Character User Interfaces for text terminals. Similarly, DSL [12] targeted UIs for only one platform: DECforms terminals displaying a 24×80 character screen. Initially, most UIDL did not necessarily wanted to support multiple platforms or operating systems since they were not many. This characterizes the first stream of multi-target UIs: the *mono-channel* aims at obtaining UIs for a single channel at a time, which is composed of a dedicated screen or terminal running a single version of an operating system.

[3] An online UIDL listing is also accessible at http://xml.coverpages.org/userInterfaceXML.html

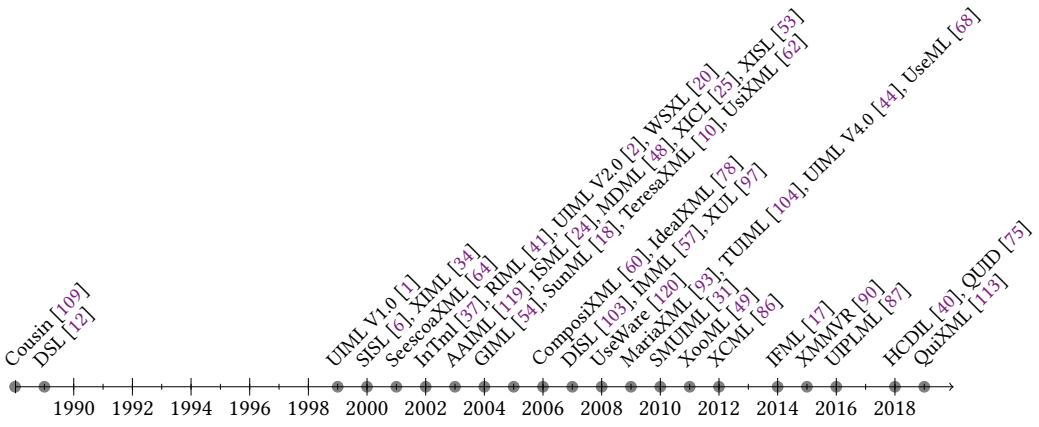

Fig. 1. Apparition of UIDLs over Time.

Then arrived the *multi-channel* [1, 29] stream aiming at obtaining UIS for one modality (always GUIs), but for different types of screens and operating systems. IN particular, WSXL targeted multi-channel UIs for workflow information systems [20]. Still in the multi-channel stream, the need for supporting multiple modalities occurred when mobile devices arrived on the market, with little screen capabilities, but with vocal capabilities. For example, XISL [53] automatically produced multimodal user interfaces running in a browser which were totally graphical in HTML, totally vocal in VoiceXML, or mixed-up in XHTML, thus allowing the end user to fill in a form by any means: graphical and/or vocal. This was particularly appreciated when the end user was able to issue a full command comprising of actions, parameters, and options at the same time instead of giving them one by one. UIDLs for multimodal applications were developed later on [59], such as and SMUIML [31] and its editor [32].

While the multi-channel continued to live, the stream of *distributed channel* found its inception when the needs for distributing parts of a UI [70] on different platforms at once were acknowledged. As opposed to the multi-channel stream where one entire UI exists for each desired target, the distributed channel divides a single UI into parts that are sent to different targets, such as different devices, screens, or platforms. Only one UI instance exists, but its parts are shuffled on different targets. From that moment, UIDL were extended to specify how different UI portions could be distributed to different screens and to manage their peer-to-peer connection.

The fourth stream of *cross-channel* appeared with the consideration of cross-device user interfaces and their corresponding UIDL, such XCML [86] or QuiXML [113]. One UI instance always exists allowing the end user to switch from one channel to another at any time. For instance, a buyer could consult an electronic product catalogue on a desktop, select an item on a tablet, and proceed to payment on a smartphone. This stream requires continuity of data and consistent interaction across the various channels: what has been done on the source platform should be transferred to the

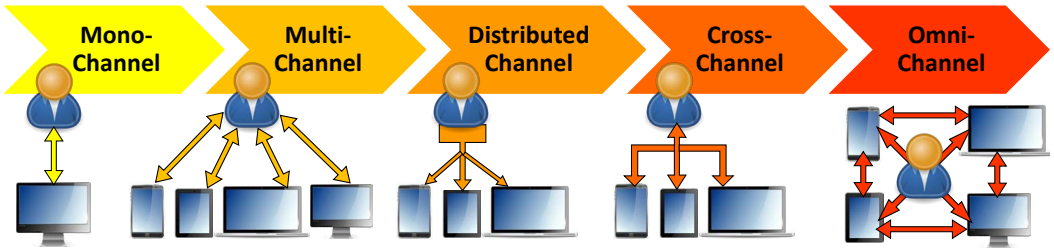

Fig. 2. Evolution of multi-target user interfaces: from mono-channel to omni-channel.

target platform to proceed. The most recent and significant UIDLs belong to this this stream, such as those UIDLs used for the W3C MB-UID recommendation [65, 96]: MariaXML [94], UseML [67], and UsiXML [62]. Updating these UIDLs require a significant development and maintenance effort, which explains why some progressively vanish, while some others are still slightly evolving [110].

Nowadays, a fifth stream appears: in the *omni-channel stream*, multiple UI instances co-exist that are adapted to each channel. In our example, browsing items of an electronic product catalogue on one channel, say a web site on a desktop, immediately updates any detailed item view running on another channel, say a native application on a tablet. A cross-channel customer would navigate through various channels to complete the same purchase while in the omni-channel stream, all channels contribute to offer a new interaction experience to end users [92].

Inspired by multi-channel experience in marketing [52] and retail [105], an omni-channel approach to customer experience collects, centralizes, and processes customer information and task data in real time. To achieve this, the different channels must work in total synergy [56, 74]. Transposed to user interfaces, an *omni-channel UI* assumes a concurrent runtime execution of multiple UI instances on different channels, whether the end user is carrying out a single task or multiple tasks. All UI instances collectively contribute to conveying a full interaction experience to the end user. The easy and seamless switching between various channels allows the end user to experience the application holistically [56]. In multi-channel, the end user picks one channel among many and completes the task in this channel. In distributed channel, the end user picks one channel that is then distributed into pieces. In cross-channel, the end user picks any channel at any time and switches from one channel to another one. In omni-channel, several channels are opened together like running several instances of cross-channel UIs.

In the next section, we will introduce OpenUIDL, a new open UIDL that fulfills these needs towards omni-channel UIs (S_8).

3 SEMANTICS AND SYNTAX OF OPENUIDL

Several UIDLs abstract UI elements, such as widgets, contents (*e.g.*, text, images, videos) by choosing a generic name to designate them. For example, UIML [44] describes the UI according to its structure, its presentation, and contents, and then employ a generic convention for each concept, like an EntryField for referring to all variations of an edit box. On the contrary, some other languages prefer not to adopt this approach and directly introduce concepts independently of their UI counterparts, like in XIML [35], where everything is defined directly at the "task & concept" level according to the CRF. OpenUIDL follows this last approach. It consists of a UIDL which supports the deployment and execution of runtime omni-channel user interfaces by fulfilling the following aims and goals:

- To describe end-user interaction: to adhere to the principle "designing interaction, not interfaces" [8], the OpenUIDL describes not only the static UI part, made up of UI elements and their relationships, but also the dynamic part, composed of user interactions, flows, events, and UI patterns based on component architectures and dynamic data driven applications.
- To generate omni-channel user interface code: the OpenUIDL generates as automatically as possible code for various tools and frameworks, while ensuring a smooth transition from one code output to another without effort.
- To enable efficient and advanced programmatic manipulation: the OpenUIDL should express abstractions enabling flexible programming techniques for integrating the resulting UI into a fully running interactive application, and not just the UI code separately.

Figure 3 reproduces a UML V2.5 Class Diagram expressing a overview of the OpenUIDL meta-model, which primarily consists of three basic constructs: OpenUIDL nodes, OpenUIDL components, and OpenUIDL project, which are further defined in the next sub-sections.

Fig. 3. An overview of the OpenUIDL semantics represented as a UML V2.5 class diagram.

3.1 OpenUIDL Nodes

The basic OpenUIDL building blocks are nodes, which always have the same description: { type: string, content: any} , where the type specifies one of the seven supported values, *i.e.*, static value, dynamic reference, element, conditional, repeat, slot, and nested-style, and the content specifies the properties of the type, which are required depending on the type (see Table 2).

A *static value* is a node holding any constant value, *e.g.*, a string, a number, or a Boolean, which will be sent as strings or numbers to the code generators: interface UIDLStaticValue { type: 'static', content: string | number | boolean } . For example, to specify a certain height of an element, this would give "height": {"type": "static", "content": "100px"} .

A *dynamic reference* is a node referencing a value that will be supplied at runtime by the application and is of three possible types: a component proposition, an internal state of a component, or a local variable. A *proposition* consists of any parameter passed by value or by reference from outside the component to be used in the rendering process. For example, <Article title="Hello…" /> and <Counter value="5" /> pass respectively one parameter by value. A proposition is represented by an object with the following properties:

- name: defines the key of the proposition, *e.g.*, "IsNatural" to refer to a proposition testing whether a number represents a natural number or not.
- value: defines the complete value of the proposition, *e.g.*, <Number.value > ="0" and Mantissa(Number.value)= "0" />.
- type: defines the data type returned by the proposition, *e.g.*, a string containing the date of the day, a number calculating the age of a person, a Boolean testing whether a number is a natural, or an object resulting from a query.
- defaultValue: specifies the default value used by the components when no proposition is provided or when it is irrelevant.
- isRequired: a Boolean flag indicating if the proposition is marked as required for future usage in the generated code.
- meta: any additional information to be passed for future usage to the code generator, *e.g.*, a suggested directive for code generation, a preference defined by the end user.

An *internal state* expresses any internal value that the component keeps privately, without exposing it to the outside world. For example, isActive, isDisabled respectively characterize whether a component like a widget will be active vs. passive and enabled vs. disabled when it is active. This would represents a push button that is displayed, but greyed because of its inactive state. A *local variable* consists of any variable local to a component that is used for internal management, such as a counter in a loop, an index in a table. Similarly to a proposition, a state definition consists of an object with the following properties:

- name: defines the key of the proposition, *e.g.*, "IsNatural" to refer to a proposition testing whether a number represents a natural number or not.
- value: defines the complete value of the proposition, *e.g.*, <Number.value > ="0" and Mantissa(Number.value)= "0" />.
- type: defines the data type of the state, *e.g.*, a Boolean indicating that a field has been filled in by the end user.
- defaultValue: specifies the initial value of the state, *e.g.*, "Unfilled" to characterize a field that was blank before any interaction.
- values: specifies an array of exact values that the state can be in, *e.g.* an entry field could be in different states such as "unfilled" in the beginning, "filling" while being used by the end user for data entry, and "completed" when the field is saturated.

The corresponding OpenUIDL specification is `interface UIDLDynamicReference { type: 'dynamic', content: { referenceType: 'prop' | 'state' | 'local' id: string } }`.

An *element* represents the most general node as it matches the concept of *Abstract Interaction Unit* (AIU) [96] specifying what information should be displayed without making any reference on how this will be achieved, and not necessarily via a graphical modality. The code generator will parse this description to apply mapping rules for selecting corresponding widgets or other UI elements belonging to the target platform. An element is decomposed into the following contents:

- elementType: specifies the AIU type, such as 'container', 'text', 'image'.
- name: specifies any custom name, with the elementType used as the default value that can be superseded at any time.
- dependency: adds information about the element in case of a custom component that is specified internally (*e.g.*, a custom widget) or coming from an external package (*e.g.*, a widget belonging to a library, like JQuery [111]).
- style: defines the visual aspect of the element in terms of attributes (key,value), where key is the name of the attribute and value is of type static, dynamic or nested-style (when styles are parsed by the code generators). Such attributes are for example properties defined in Cascading Style Sheets (CSS) which could be exploited in dialog modeling [115].
- attrs: defines any custom attribute that is specific for this element. These attributes will be translated into dynamic values inside the code generated.
- events: defines a list of instructions to be added on event handlers. For instance, if the end user clicks on a pushbutton, a certain semantic function should be triggered.
- children: consists of a hierarchy of optional nodes that refine the element. For instance, a pushbutton could consist of an icon and a label. By using this mechanism, UI composite patterns could be defined to be reused elsewhere, for example a particular pushbutton having a reserved label, keyword, shortcut, and icon.

A *conditional* node is a node used when a conditional expression should be specified. For example, if the end user has entered some data in a fled, then a reaction should be triggered. The content node contains:

- node: the node instance placed behind the conditional node.
- reference: a dynamic reference value based on which the rendering condition is working.
- value: the value of the dynamic reference for which the node is displayed.
- condition: the explicit conditional expression based on which the node is displayed.

For example, a conditional node could test the equality between a value specified statically and a dynamically provided value: `reference > 3`. Any unary operator, *e.g.*, increasing or decreasing a value by 1, or any binary operator, *e.g.*, addition, subtraction, can be used similarly.

The Repeat node specifies a repetitive structure over a certain node, like repeating an item in a list, repeating a cell in a table or a matrix. The content allows the following fields:

- node: the UIDLNode that will be placed inside the repeater, such as a cell in an array.
- dataSource: the data structure holding the raw data over which the structure should iterate, such as a complete array of values.
- meta.useIndex: the optional index declared as the position of the element in the structure, such as a pointer in the array.
- meta.iteratorName: a string which overrides the name of the variable inside the iteration (per default, an item), such as the name of the cell in the array.
- meta.dataSourceIdentifier: a string which identifies the local data source variable inside the component, *e.g.*, when a static array is passed as a dataSource and the framework needs to declare that array as a local variable.

	Root node	Child element	Attribute	Style	Conditional reference	Attribute source
Static value	✓	✓	✓	✓		✓
Dynamic reference	✓	✓	✓	✓	✓	✓
Element node	✓	✓				
Conditional node	✓	✓				
Repeat node	✓	✓				
Slot node		✓				
Nested style node				✓		

Table 2. Contents appropriate for the seven types of nodes.

The Slot node allows defining a placeholder in a list of components that will be determined at runtime. For example, an array of items will consist of items coming from a database and from a list enterd by the end user at runtime.

The nested-style node specifies a list of style properties that can be applied at any level of the component hierarchy. For example, a nested-style hold CSS properties that are applied directly on the root node of a component and throughout its definition. Responsive styles can be defined by adding a nested-style at a certain level of the hierarchy to alter the layout that is statically defined at a higher level, thus allowing the superseding of the highly defined style by any local style activated based on conditions.

3.2 OpenUIDL Components

By definition, a OpenUIDL component specifies a hierarchy of OpenUIDL Nodes, together with some top level declarations used to identify the dynamic data inside the hierarchy. A component is described through the following attributes:

- name: a unique string name used to identify the component. This name is used for creating the component name, but also the file name when used in a project generation process.
- node: any instance of OpenUIDLNode specified as the root node of the component.
- schema: an optional Unified Resource Locator (URL) pointing to the schema corresponding to this component.
- meta: an optional object containing dynamic values.
- stateDefinitions: an optional object containing information defining the states of a component. Basically, a state definition is modelled as a pair (attribute, value), a data structure widely used in feature modelling on the Web [95], where the attribute represents the key of the attribute and value represents the current value of this key. To completely represent a state, we augment this definition with the following attributes: stateType representing the data type (*i.e.*, Boolean, number, string), defaultStateValue representing the default value of the attribute if any, potentialStateValues representing the runtime potential values of the attribute, and metaValue representing any meta-information about this concept. For instance, a list of possible zip codes of a country could be modelled by stateDefinitions holding the possible zip codes and the current one depending on the user's location.
- propDefinitions: an optional object with information used as a content for the component. These definitions contain a list of propositions, as defined above, with the same data structure.

3.3 OpenUIDL Project

Now that the building blocks of OpenUIDL are defined, they can be composed to form a OpenUIDL model that is captured in a OpenUIDL project. We hereby define a OpenUIDL project as hierarchy of components augment with some information on top of it, related to global settings, assets, and routing. Therefore, its corresponding structure is as follows:

- name: a unique project name representing the identifier of the project.
- root: the root component of the hierarchy of components that is considered to be the entry point of the project.
- globals: an object containing project related information.
- schema: an URL pointing to the current version of the project schema.
- components: the hierarchy of OpenUIDL components.

In the above structure, the globals node contains information related to the project as follows:

- settings: an object containing the settings that are global to the project, such as supported languages and titles.
- assets: an array of assets, where an asset is represented as an attribute-value pair [95] to capture project resources. For example, the attribute specifies the resource type, *e.g.*., a style, a script, an icon, a font) and the value specifies either the real value of the asset or the path to the location where it is defined. For example, an interactive application could contain a menu bar with menu items, some of them being reserved (*e.g.*, "File", "Edit"), some others being specific (*e.g.*, "Assess Loan"). The reserved keywords are usually stored on web site for localization and internationalization (in this case, the path is known) or stored locally (in this case, the translated values are stored locally).
- meta: an array of objects containing any other information requested by the project to properly act at runtime. For example, a description, a list of keywords, viewports, a list of supported screen resolutions.
- manifest: an object containing the information for the manifest description, which is hereby referred to as the metadata for a group of accompanying files or resources that are related to each other [47]. For example, the creation date of the project, the owner, the last modification date, the version number, and the license.

3.4 Syntax of OpenUIDL

Defining the semantics of a language is usually conducted through the practice of meta-modelling, which results into a formal way for defining an abstract syntax of a domain specific modelling language, such as OpenUIDL. Section 3 defined the semantics of OpenUIDL by providing a UML Class diagram, one of the most frequently used methods for this purpose, but not the only one. The semantics, sometimes referred to as the abstract syntax, should be clearly separated from any concrete syntax implementing it.

Nowadays, no universally method has been promoted for formalizing the definition of concrete syntax. The very first UIDLs saw their syntax defined by a context-free grammar, such as Cousin [109], or by an Extended Backus-Naur Form (EBNF) [117], such as DSL [12]. When SGML and XML languages appeared, in addition to problems raised by the usage of EBNF [117], soon were UIDLs defined according to a XML Schema, such as UIML [1] and XIML [34]. The vast majority of subsequent UIDLs adopted a XML-compliant notation [107], most of them based on a publicly available XML schema, but not systematically. Some UIDLs never released their XML schema, thus preventing any interested party to properly use it or to extend it. The most recent UIDLs, such as QUID [75], tend to define their syntax based on a JSON (JavaScript Object Notation) file [46], a commonly used lightweight data-interchange format. This format is expected to be relatively

Fig. 4. A simple "HelloWorld" user interface.

easy for both humans to read and write and for machines to parse and process, such as for code generation. While these quality properties may be not very important for other Domain Specific Languages, they are critical for UIDLs as they may be read by human being to understand how a UI is modelled.

On the other hand, TypeScript interface receives today more attention in that it promotes a structure that defines the contract to be satisfied by an interactive application [7]. It defines the syntax for classes to follow. Classes that are derived from an interface must follow the structure provided by their interface. Since the semantics of OpenUIDL are defined via a class diagram, a natural transition from the abstract syntax to a concrete syntax is offered by TypeScript interfaces. Such a TypeScript interface is not necessarily bound to a particular language since a TypeScript compiler does not convert interface into code like JavaScript. It merely uses interface for type checking at run-time.

For these reasons, we chose to define the syntax of OpenUIDL by both a JSON and a TypeScript interface formats. Appendix A, respectively Appendix B, reproduce the JSON definition for the OpenUIDL Component, respectively the Project, concepts. Appendix C reproduces the TypeScript interface for the OpenUIDL, thus demonstrating that multiple syntaxes can be exploited starting from the same semantics.

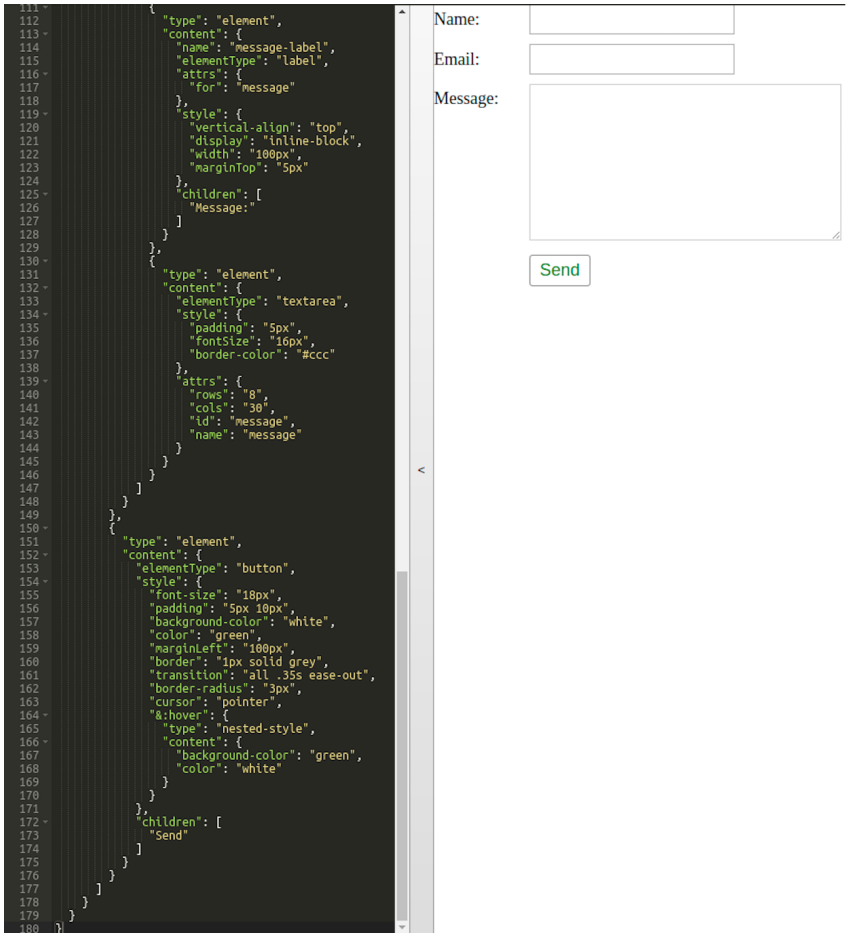

Fig. 5. A "Contact Form" user interface.

3.5 Examples of Using OpenUIDL

Fig. 4 reproduces the OpenUIDL specifications of the classical "HelloWorld" user interface. In this project, a simple component is specified with a heading, having "Hello" as a default value, and containing only one element of type "container". This container itself contains a single element of type text with the capability to either display the default value "World!" (represented by a static value) or a dynamic value (represented here as a proposition calling for the heading of the user interface). The concatenation of the two strings produces the final result.

Fig. 5 reproduces the OpenUIDL specifications of a simple "Contact form". In this project, a more complex component is decomposed into a container and an element of type "button" to represent the "Send" push button. The container is recursively decomposed into three aligned lines, each line containing its label that are all left aligned and their corresponding elements, also left aligned. The last widget is of type "textArea". In principle, this structure is sufficient to produce a user interface. Once a style is applied, specifications of these elements are complemented with additional properties that are automatically provided with the style definition, such as the colors of the borders, the font size, etc. Changing the size will only affect the stylistic specifications of these attributes, without changing their structure.

4 MODEL-BASED APPROACH WITH OPENUIDL

The goal of this section is not to introduce yet another software development methodology that integrates UI engineering with the process, but instead to show the steps of the proposed model-based approach for omni-channel UIs. Therefore, the approach for devising the method is more descriptive than prescriptive, with a primary goal to preserve the process of existing methods, while tackling the challenges introduced by the contemporary omni-channel scenario. Finally, the method is presented as independent of technological support, which is then added in order to support the assessed requirements.

By definition, a *method* is a particular form of procedure for accomplishing or approaching something, especially a systematic or established one. In the case of this paper, the method is a systematic, step-wise approach for producing omni-channel user interfaces.

The Software Process Metamodel (SPEM) [91] notation will be used to describe our step-wise approach for the following reasons. SPEM is an OMG specification of an UML metamodel and UML profile and is widely used to represent a family of software development processes and their components. It constitutes a sort of "ontology" of software development processes, providing the minimum set of process modeling elements to describe any software development process without adding specific models or constraints for any specific area or discipline, such as software engineering, requirements engineering or project management. We will use the following sub-set of SPEM concepts, along with their corresponding standardized icons (Fig. 6):

- Work Definition: constitutes a kind of operation that describes the work performed in the process.
- Work Product: constitutes any tangible piece of information that is produced, consumed or modified by a process.
- Process Role: defines responsibilities over specific Work Products.
- Activity: constitutes a piece of work performed by a single Process Role.
- Document: constitutes a special kind (a stereotype) of Work Product.
- Discipline: partitions the Activities within a process according to a common theme.
- Guidelines: constitutes an element aimed at providing more detailed information about some resource.

Fig. 6. SPEM concepts used in this section with their icons.

Based on these concepts, Fig. 7 depicts a conceptual model expressing how they are semantically related to each other. Different roles, like a designer or a end user, could be played to produce one or several prototypes, whose user interface is a product obtained by applying various techniques. The prototypes are assumed to be adapted to one or many devices, a device being considered as a specific type of a computing platform.

The abstractions of method roles comprise [66] *designers*, who are those who must "tell" end users what they mean by the artifact they have created, *developers*, who are those who need to program the artefact designed by the designer, and *end users*, who are those who are expected to understand and respond to the "message" from the designer. Therefore, the Process Roles considered for the method are:

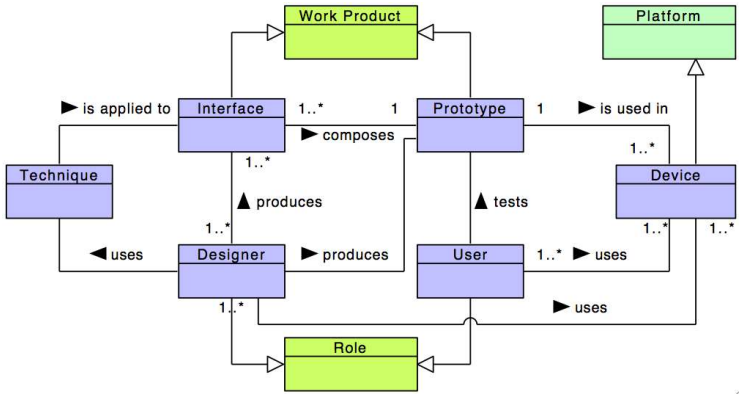

Fig. 7. Conceptual model of the major SPEM concepts.

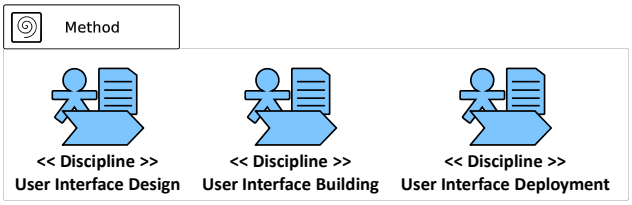

Fig. 8. The method defined as a SPEM package with three basic disciplines.

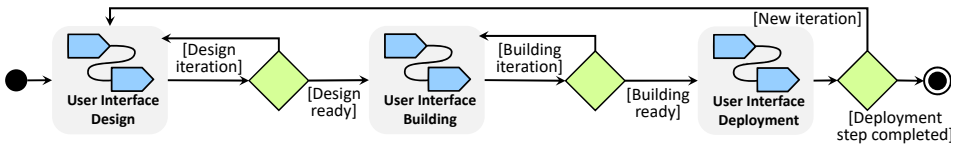

Fig. 9. Method workflow defined as a SPEM activity diagram.

- *Designer*: this role is responsible for producing early or detailed designs of UIs and designing the navigation independently of the target platforms. It is also responsible for conducting the design session and, when is the case, performing user tests with the produced prototypes.
- *Developer*: this role is responsible for developing the code corresponding to the design artefact by the designer. It is also responsible for conducting the development session.
- *End User*: this role is responsible for providing feedback throughout the design session, both by helping with designing the UIs and also by validating them.

The OpenUIDL method is decomposed into three successive disciplines (Fig. 8):

(1) In the user interface design, the designer is responsible for conducting the early and detailed design, preferably with the collaboration of end users or their representative to support participatory design [13].
(2) In the user interface building, the developer is responsible for building the complete project from the design specified by the designer in OpenUIDL, ready for deployment.
(3) The developer is responsible for deploying the complete project on the server to obtain the runtime application [11] with its omni-channel UI, to be delivered to the end user.

After completing each discipline, a checkpoint could occur to validate its results or, instead, to go through a loop to support iterative design (Fig. 9). These three disciplines are further detailed in the next sub-sections.

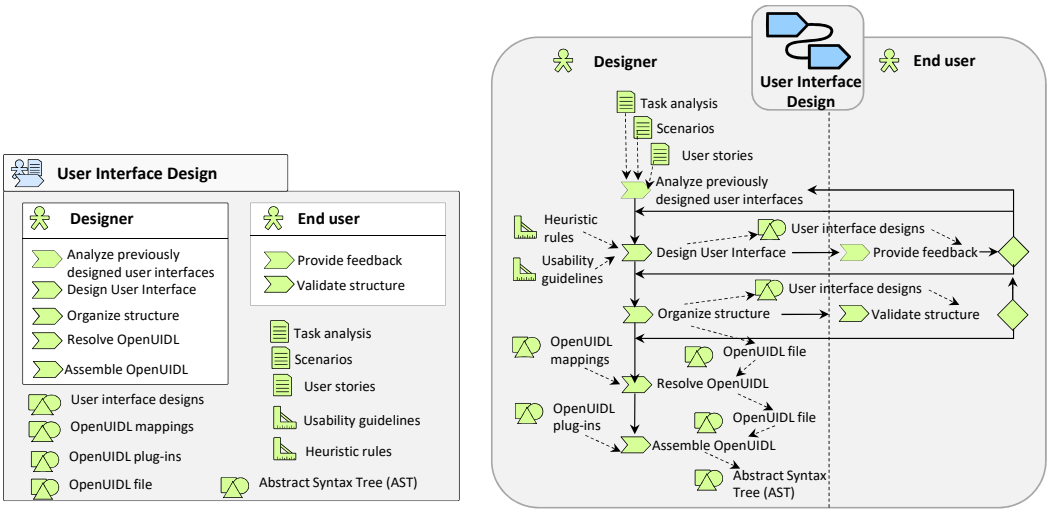

Fig. 10. Discipline 1 SPEM definition: (a) discipline package, (b) activity diagram.

4.1 Discipline 1: User Interface Design

Fig. 10 depicts the first discipline of the method used for model-based approach of omni-channel UIs with OpenUIDL. The designer analyzes previously designed user interfaces, whether they exist in the same domain or not, to foster reusability, as demonstrated by Delgado *et al.* [27]. This consideration includes various documents produced during the requirement engineering and/or task analysis: a task analysis (which recursively decomposes the end user's tasks into a hierarchy of sub-tasks to end up with actions according to W3C recommendation [96]), a set of user scenarios (which represent instantiations of the task analysis presented as a designer-oriented narration of how actions are ordered in time and space from the task analysis [88]), or a set of user stories (which represent a suite of structured sentences written by the end user herself [22] to characterize how a particular task could be carried out in a particular context of use [19]). In practice, there are many techniques that can be used for eliciting requirements from stakeholders [88], such as interviews, use cases, scenarios, user-task elicitation, user stories, observation and social analysis, focus groups, brainstorming sessions, and prototyping.

The designer then creates a UI project within the Playground (Fig. 11), a cloud-based publicly available graphical editor, and further edits the project, retrieves any project from the database, deletes any, and searches for any. Many UIDLs provide a stand-alone UIDL editor, such as GrafiXML [72] for UsiXML [62], IdealXML [78], which requires a particular setup, a parameterization to be integrated in a software suite, and other calibration, all these activities being time consuming and difficult to achieve. QUID [75, 76] is probably an exception as its editor is running in a web page. In contrast, the Playground is a visual editor that relieves any designer from such non-design oriented activities by offering a cloud-based, always accessible editor, in which any widget can be dragged from a palette and dropped onto a design canvas. While the designer is editing a UI, the Playground automatically generates and maintains the corresponding OpenUIDL specifications in a separate window that can be displayed. For the sake of the design process, there is no need to see or edit the OpenUIDL specifications unless the designer really wants to. In this case, the Playground maintains a two-view approach in which each modification brought to one view is automatically propagated in the other view. This fashion allows copying any portion of OpenUIDL specifications from another project and pasting it to the current one without requiring to re-layout objects, as they are relocated as

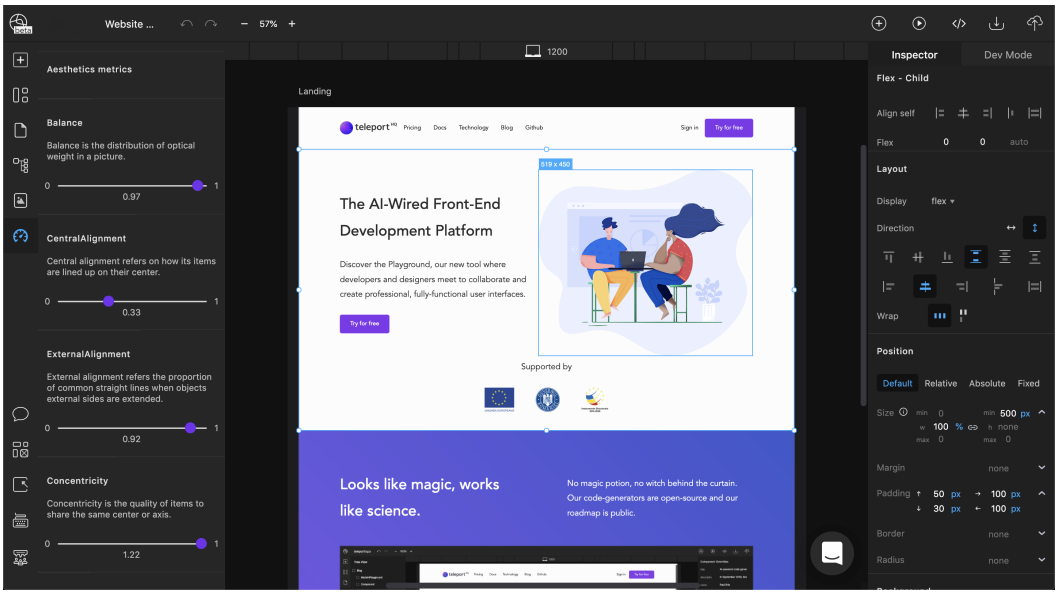

Fig. 11. The Playground cloud-based on-line editor for OpenUIDL, accessible at https://play.teleporthq.io/

.

in [98]. The user interface designs are submitted to the end user for feedback by providing the project URL to support iterative design. The designer then refines the UI structure according to the structure of nodes offered by the OpenUIDL until this structure is officially validated by the end user or her representative. The designer then completes this discipline by performing two activities.

First, the OpenUIDL project is subject to the mapping problem [78]: a set of mappings is applied to the OpenUIDL project for any target decided for the omni-channel UI, as in [23]. A *mapping* consists in transforming a OpenUIDL node into one counterpart (one-to-one mapping). For example, a element is always mapped onto an edit field in HTML, a container onto a div section, a text onto a span tag. Mappings are specified in an editable mapping file consisting of a JSON structure that translates OpenUIDL elements, attributes, and events into technology specific values. Appendix D reproduces the definition of a mapping file for OpenUIDL to HTML. Since this file is editable, multiple generation configurations can be supported, such as more sophisticated rules for selecting widgets or for layout (as in XIML [35]) and MoCaDix [113]). When multiple mappings are applied, only the last one prevails. For example, if the mapping OpenUIDL to HTML is applied before the mapping OpenUIDL to React, the elements shared by both languages will be saved and taken from React. This mechanism for mapping propagation offers even more flexibility in the variation of UIs that can be generated.

A set of mappings exists for each channel such as: OpenUIDL to HTML5, OpenUIDL to React.js, OpenUIDL to Vue.js, etc.[4] Selecting the target immediately generates the corresponding code for the target envisioned, an activity that can be repeated for any target. Again, the results of the mappings can be displayed in a separate coordinated window for design preview, although it is not intended to require from the designer to tweak its contents or to intervene.

[4]The accompanying video Examples.mp4 demonstrates the application of mapping files for different channels: SimpleComponent illustrates the automated UI generation for multiple channels for the simple "Hello World" example initiated in Fig. 4, ComplexComponent illustrates the same process for a more advanced example, and ContactForm illustrates it for the "Contact Form" case study initiated in Fig. 5. The NavBar example shows how to specify a menu bar with page navigation.

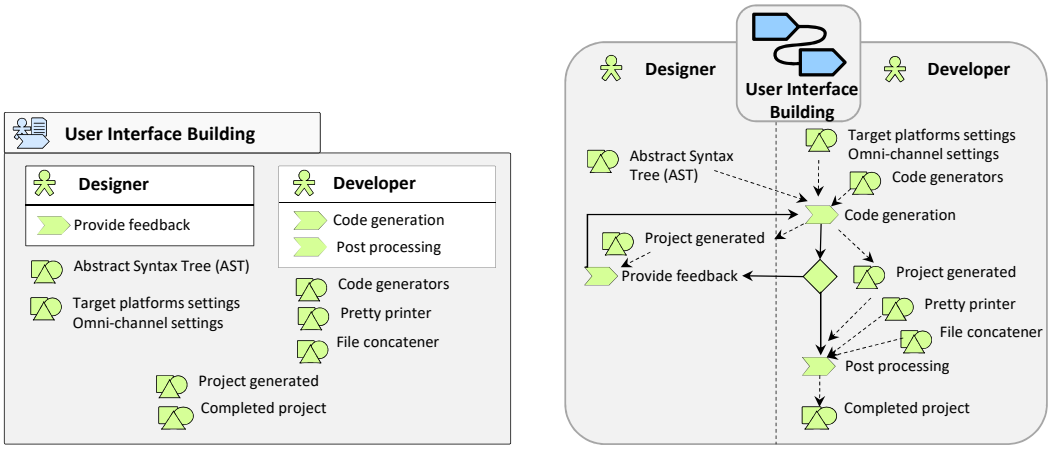

Fig. 12. Discipline 2 SPEM definition: (a) discipline package, (b) activity diagram.

Second, define the style by assembling the OpenUIDL with style plug-ins, such as for Cascading Style Sheets (CSS3), React.js styles, Import, etc. While the OpenUIDL specifications are maintained, this discipline results into the automatic creation of an Abstract Syntax Tree (AST), which serves as an input to the second discipline. There are multiple plug-ins per framework based on the role played by a plug-in, such as for style only, for contents only, or for both. For example, a React-base plug-in creates the AST of the main part of the component, a react-css plug-in handles the styling, a proptypes plug-in adds type safety to React components, and an import plug-in for handling the dependencies of a file. All ASTs are stored in memory throughout the life cycle of this discipline.

An *Abstract Syntax Tree* (AST) [85] represents any abstract syntactic structure of source code written in a programming language or descriptions written in a markup language. Each node of the tree denotes a construct occurring in the source file and nodes are connected with an operator that uses them as input/output. We used ASTs in the context of omni-channel UIs for the following reasons: both the OpenUIDL describing it and any source code or markup language can be expressed by an AST, for the structure and the style, transforming a source AST in OpenUIDL into a target AST for a particular language can be achieved by graph grammars and tree transformations in a rigorous way [114], thus enabling the developer to follow code evolution [85] and to reason about a UI at any time [106], such as for forward engineering or reverse engineering [15].

A project generator converts a Project UIDL into an in-memory structure of files and folders. The core package that implements the project generation algorithm is generic-project-generator.

4.2 Discipline 2: User Interface Building

Fig. 12 depicts the second discipline of the method used for model-based approach. The AST exported from the previous discipline by the designer is sent to the developer for performing the code generation according to settings for the target platforms (*e.g.*, screen resolution, operating system) and omni-channel settings (*e.g.*, specifications of OpenUIDL sections that can be synchronized on different targets), based on Model-to-Code (M2C) generators (*e.g.*, Babel AST generator for the JavaScript code[5], JSS compiler) and Model-to-Markup (M2M) producers (*e.g.*, HAST-to-HTML[6] automatically generates an AST from HTML code based on the Hypertext Abstract Syntax Tree format). For the moment, code generators are implemented for the following web UI channels:

[5]See https://astexplorer.net/
[6]See (https://github.com/syntax-tree/hast)

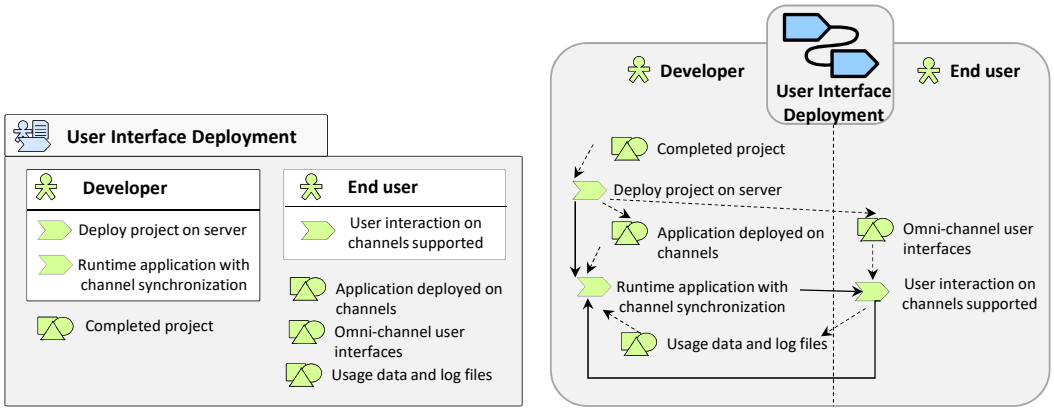

Fig. 13. Discipline 3 SPEM definition: (a) discipline package, (b) activity diagram.

- React.js[7] [14]: a JavaScript library for building graphical user interfaces of either Single-Page interactive application (SPA) or for mobile user interfaces.
- Vue.js[8] [30]: an open-source Model–view–viewmodel (MVVM) JavaScript framework for building graphical user interfaces for Single-Page interactive applications. Vue components extend basic HTML elements to encapsulate reusable code [116]. Components are custom elements to which the Vue's compiler attaches some interactive behavior.
- Preact[9] [108]: a JavaScript lightweight library improving React with the same ES6 API.
- Stencil[10] [5]: a toolchain for building reusable, scalable web applications based on Web Components that run in every browser.
- Angular[11] [81]: a JavaScript-based open-source front-end web framework for client-side model–view–controller (MVC) and model–view–viewmodel (MVVM) architectures, along with components commonly used in rich Internet applications. It is also considered as the front-end layer of the MEAN stack.
- ReactNative[12] [33]: an open-source mobile application framework created for developing user interfaces for multiple targets, such as Android, iOS, Web, and UWP.

Each code generator generates one or many UIs from the same UIDL depending on the UI instances required at run-time by the application (see accompanying video Examples.mp4. In particular, ReactNative is efficient in that it generates native code for multiple targets at once. The project generated is returned to the designer for further checking and validation before post processing: a pretty printer reformats source code and markup language and a file concatener appends files for producing code documentation of the completed project.

4.3 Discipline 3: User Interface Deployment

Fig. 13 depicts the third discipline of the method used for model-based approach of omni-channel UIs with OpenUIDL. The project completed in the previous discipline is deployed by the developer on the target server according to a software architecture pattern, like a MEAN or MERN stack. The MEAN software architecture has the following benefits [45]: (1) all MEAN stack components

[7]See https://reactjs.org/
[8]See https://vuejs.org/
[9]See https://preactjs.com/
[10]See https://stenciljs.com/
[11]See angularjs.org
[12]See https://facebook.github.io/react-native/

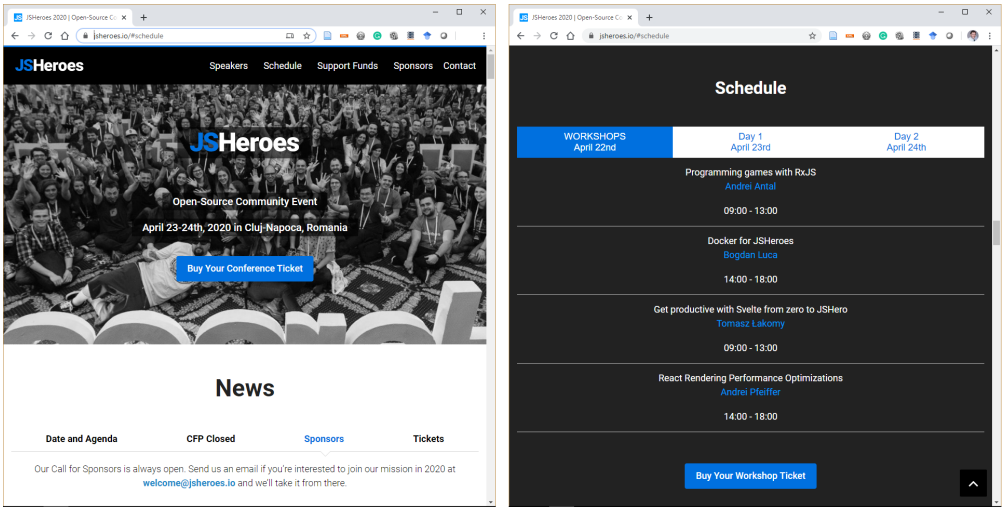

Fig. 14. A real-world web site deployed at https://jsheroes.io: (a) a static front page, (b) a dynamic schedule.

are open-source and free to use, (2) MEAN is straightforward to use for both back- and front-end as opposed to using different languages and technologies for front and back ends; (3) only one language for both server-side and client-side execution is developed; (4) it enables working on the front-end and back-end of the application using common languages (JavaScript and TypeScript, respectively), as well as to stay flexible with the diverse middleware and libraries.

Instead of deploying an interactive application on a local server, the OpenUIDL project, along with its generated files, can be submitted to *web application publishers*, which fall into two categories:

(1) *Direct publishers*: are organisations receiving as input the whole OpenUIDL project along with the generated files and folders for creating the final building and deployment of the application hosted on their infrastructure. As output, a public link gives access to the web application. For example, Netlify[13], a cloud computing company offering hosting and serverless backend services for websites belongs, to this category. Similarly, a OpenUIDL project can be sent to http://www.zeit.com, which automatically builds the application with a URL on their site.
(2) *Indirect publishers*: are organisations or services hosting the complete OpenUIDL project and its generated files and folders, host them in different forms, *e.g.*, as a .ZIP file to be downloaded locally, and deploy them while the project is still evolving. For example, the Write-to-Disk mini-application writes the complete code from any process to a disk to execute some library (*e.g.* Node.js, a GitHub repository or a CodeSandbox[14] publisher, an on-line code editor and prototyping tool that enables to upload the generated OpenUIDL code and run it in a browser.

Once deployed, the run-time application is then called through its URL by the end user within her Internet browser. Several instances of this browser could be opened at once on the same target platform or on different ones, asynchronously or synchronously. Each time a page is requested by the end user whatever the platform is, the server sends the page corresponding to the target platform, even several different ones at once. The accompanying video Design, Develop, and Deploy.mov illustrates some significant steps in the process of designing static and dynamic pages, developing its corresponding code in React.js to obtain the real-world web site deployed at https://jsheroes.io/ (Fig. 14). Its OpenUIDL specifications with HTML and Vue code are accompanying this paper.

[13] See https://www.netlify.com/
[14] See https://codesandbox.io/

5 DISCUSSION

In order to assess the potential benefits of OpenUIDL as it is supported by its IDE, we revisit the eight shortcomings stated in the introduction in the light of the current implementation.

S_1. A *limited accessibility of their definition*: contrarily to some UIDLs without semantics and syntax publicly available for different reasons, *e.g.*, for royalties, OpenUIDL is open, entirely documented (see Section 3) on-line and publicly available at https://docs.teleporthq.io/guides/what-is-this.html, where more examples are explained. This requirement is therefore considered satisfied.

S_2. A *loose integration into code generators*: contrarily to some UIDLs involving a UI model editor that is separate from code generators or transformers, the Playground editor for OpenUIDL (Fig. 11) is also publicly available at https://play.teleporthq.io/signin as well as the code generators tightly integrated (see Figs. 4, 5, and videos accessible at https://presentation-website-assets.teleporthq.io/teleport-video.mp4). A YouTube channel is also accessible at https://www.youtube.com/channel/UC_hv653G9B7gsVjMQ6DuSgA. Therefore, the transition from modelling to code generation is seamless. Any modification in the OpenUIDL model is automatically propagated in the generation. This requirement is therefore fulfilled.

S_3. A *narrow generation bandwidth*: contrarily to some UIDLs focusing on only one target, OpenUIDL is supported by several code generators (see Section 4.2): React.js, Vue.js, Preact, Stencil, Angular, and ReactNative. Of course, such a list will be never complete and will evolve over time with the capabilities of these target environments. The OpenUIDL open source project is accessible on a GitHub repository at https://github.com/teleporthq. More specifically, code generators are publicly available at https://github.com/teleporthq/teleport-code-generators. In this work, we target omni-channel UIs, thus explaining why only web based applications are concerned for their run-time capabilities. The OpenUIDL could accommodate in principle other code generators, such as for imperative languages (*e.g.*, Java or C#) or functional (*e.g.*, LISP or Prolog). We however believe that web applications represent the major targets, as revealed by the latest advances in the domain [76]. Consequently, this requirement is partially fulfilled.

S_4. A *limited language expressiveness*: the language expressiveness is always at the price of introducing new specifications. Like other UIDLs, OpenUIDL does not escape from this dilemma as it promotes 17 first-class concepts (see Fig. 3) that are not always straightforward to master. Fortunately, all these tags are automatically generated by the the Playground editor (Fig. 11 - see also https://play.teleporthq.io/) and later transformed into code (see Section 4.2). What is however appealing is that there is no need to entirely specify all aspects of a OpenUIDL element or component at once: an element can be progressively specified along with the level of details of the project. Consequently, this requirement is partially satisfied, in a fair and reasonable way.

S_5. A *limited support for run-time user interface*. Several OpenUIDL mechanisms, (Section 3) support dynamic user interfaces, namely by filling contents at run-time, by retrieving data from a database at run-time, by evaluating propositions at run-time, by specifying adaptive elements and components based on their propositions, such as for navigation within or across web pages. However, OpenUIDL will never offer a run-time support that would be equal to manual programming or to an adaptation engine [34] that is offered by some environments [94], run-time engines [11], such as for adaptive dialog [41], adaptive layout [98], and adaptive accessibility [39]. All these types of adaptation represent significant efforts by themselves. Therefore, we believe this this requirement is minimally addressed.

S_6. A *limited integration into the rest of the interactive application*: Contrarily to some UIDLs which only generate UI code, a OpenUIDL project covers an entire interaction project that goes beyond the mere description of a static UI [8]. An entire project can be specified and give rise to a fully running application (see Section 4.2). This requirement is satisfied.

S_7. A *lack of efficient software support*: contrarily to some UIDLs shipped only with their editor, OpenUIDL works hand in hand with its Playground editor (see Fig. 11) and development environment that is cloud-based. Thus, we believe that this requirement is fulfilled depending on the capabilities offered by the software suite released for this purpose, which is adequate.

S_8. *No support for omni-channel user interfaces*: contrarily to some UIDLs offering support for one channel (*e.g.*, say HTML4) or for multiple channels or for cross-channels [86], OpenUIDL enables the designer to produce omni-channel UIs by generating as many UI targets as required by the various contexts of use that could be run concurrently or synchronously, on different devices simultaneously or not. Yet, we believe that this requirement is still in its infancy due to the recency of the concept. Indeed, we do not have (yet) a model for characterizing which parts of a OpenUIDL project could be conveyed separately or together, so as to create a real omni-channel experience as it exists in other domains, like marketing. The various channels are selected, produced, and run concurrently, by the same user or not. It is therefore a one-or-nothing constraint that could be relieved by exploiting a channel model.

6 CONCLUSION AND FUTURE WORK

This paper first presented the new concept of omni-channel user interfaces (S_8), one step beyond cross-channel user interfaces, namely by distinguishing it from other types of multi-target user interfaces. For this purpose, the paper defined the semantics (based on a UML V2.5 Class diagram) and the syntax (based on JSON and TypeScript formats) of OpenUIDL, an open, publicly available (S_1), user interface description language for describing (S_4) runtime (S_5) omni-channel user interfaces (S_8), based on its semantics and its syntax. Then, the model-based approach for developing UIs based on OpenUIDL has been decomposed into three successive disciplines (S_6): the user interface design, the user interface building, and the user interface deployment based on six available code generators (S_2) for six targets. Each discipline has been systematically described in terms of the SPEM notation and illustrated on some examples. Since the Playground editor for OpenUIDL, along with its development environment and its corresponding code generators, are accessible as a cloud-based environment, there is no setup required and all actions can be traced into log files that can be undone or replayed. To understand how designers and developers will be using this all-in-one, cloud-based environment, for their project, with their consent, we would like in the near future to analyze the log files storing their activity in order to optimize the environment.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments on earlier versions of this manuscript.

REFERENCES

[1] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. 1999. UIML: an Appliance-independent XML User Interface Language. *Computer Networks* 31, 11 (1999), 1695–1708. https://doi.org/10.1016/S1389-1286(99)00044-4

[2] Mir Farooq Ali, Manuel A. Pérez-Quiñones, Marc Abrams, and Eric Shell. 2002. *Building Multi-Platform User Interfaces with UIML.* Springer Netherlands, Dordrecht, 255–266. https://doi.org/10.1007/978-94-010-0421-3_22

[3] Nathalie Aquino, Jean Vanderdonckt, Nelly Condori-Fernández, Óscar Dieste, and Óscar Pastor. 2010. Usability Evaluation of Multi-device/Platform User Interfaces Generated by Model-driven Engineering. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (September 16-17, 2010)

(ESEM '10). ACM, New York, NY, USA, Article 30, 10 pages. https://doi.org/10.1145/1852786.1852826 See experiment web site at http://https://f51srniiblzzphhelwqxvq-on.drv.tw/naweb/mdp-usability-eval/.

[4] Nathalie Aquino, Jean Vanderdonckt, and Oscar Pastor. 2010. Transformation Templates: Adding Flexibility to Model-driven Engineering of User Interfaces. In *Proceedings of the ACM International Symposium on Applied Computing* (March 22-26, 2010) *(SAC '10)*. ACM, New York, NY, USA, 1195–1202. https://doi.org/10.1145/1774088.1774340

[5] Michael McCool Arch Robison, James Reinders. 2012. *Structured Parallel Programming*. Morgan Kaufmann, San Francisco, United States. https://www.oreilly.com/library/view/structured-parallel-programming/9780124159938/

[6] Thomas Ball, Christopher Colby, Peter Danielsen, Lalita Jategaonkar Jagadeesan, Radha Jagadeesan, Konstantin Läufer, Peter Mataga, and Kenneth Rehor. 2000. Sisl: Several Interfaces, Single Logic. *International Journal of Speech Technology* 3 (6 2000), 93–108. https://doi.org/10.1023/A:1009645414233

[7] Thomas Ball, Peli de Halleux, and Michał Moskal. 2019. Static TypeScript: An Implementation of a Static Compiler for the TypeScript Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '19)*. Association for Computing Machinery, New York, NY, USA, 105–116. https://doi.org/10.1145/3357390.3361032

[8] Michel Beaudouin-Lafon. 2004. Designing Interaction, Not Interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '04)*. Association for Computing Machinery, New York, NY, USA, 15–22. https://doi.org/10.1145/989863.989865

[9] Abdo Beirekdar, Jean Vanderdonckt, and Monique Noirhomme-Fraiture. 2002. *A Framework and a Language for Usability Automatic Evaluation of Web Sites by Static Analysis of HTML Source Code*. Springer Netherlands, Dordrecht, 337–348. https://doi.org/10.1007/978-94-010-0421-3_29

[10] Silvia Berti, Francesco Correani, Giulio Mori, Fabio Paternò, and Carmen Santoro. 2004. TERESA: A Transformation-based Environment for Designing and Developing Multi-device Interfaces. In *Proceedings of ACM International Conference on Human Factors in Computing Systems, Extended Abstracts (CHI EA '04)*. ACM, New York, NY, USA, 793–794. https://doi.org/10.1145/985921.985939

[11] G. Blair, N. Bencomo, and R. B. France. 2009. Models@run.time. *IEEE Computer* 42, 10 (Oct 2009), 22–27. https://doi.org/10.1109/MC.2009.326

[12] François Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, Isabelle Provot, and Jean Vanderdonckt. 1989. Model-based user interface design in Trident with Dynamic Specification Language (DSL). In *Proceedings of the IFIP WG 2.7 Working Conference* (September 1989). Presses Universitaires de Namur, Namur.

[13] Susanne Bødker and Morten Kyng. 2018. Participatory Design That Matters - Facing the Big Issues. *ACM Transactions on Computer-Human Interaction* 25, 1 (Feb. 2018), 31. https://doi.org/10.1145/3152421

[14] Adam Boduch. 2019. *React Material-UI Cookbook*. Packt Publishing, Birmingham, United Kingdom. Accessible at https://www.packtpub.com/application-development/react-material-ui-cookbook.

[15] Laurent Bouillon, Quentin Limbourg, Jean Vanderdonckt, and Benjamin Michotte. 2005. Reverse engineering of Web pages based on derivations and transformations. In *Proceedings of the IEEE Third Latin American Web Congress* (October 31–November 2, 2005) *(LA-WEB '05))*. 11 pp. https://doi.org/10.1109/LAWEB.2005.29

[16] Laurent Bouillon, Jean Vanderdonckt, and Jacob Eisenstein. 2002. Model-Based Approaches to Reengineering Web Pages. In *Proceedings of the First International Workshop on Task Models and Diagrams for User Interface Design* (July 18-19, 2002) *(TAMODIA '02)*, Costin Pribeanu and Jean Vanderdonckt (Eds.). INFOREC Publishing House Bucharest, 86–95.

[17] Marco Brambilla, Andrea Mauri, and Eric Umuhoza. 2014. Extending the Interaction Flow Modeling Language (IFML) for Model Driven Development of Mobile Applications Front End. In *Mobile Web Information Systems*, Irfan Awan, Muhammad Younas, Xavier Franch, and Carme Quer (Eds.). Springer International Publishing, Cham, 176–191.

[18] Christian Brel, Philippe Renevier-Gonin, Alain Giboin, Michel Riveill, and Anne-Marie Dery. 2014. Reusing and Combining UI, Task and Software Component Models to Compose New Applications. In *Proceedings of the 28th International BCS Human-Computer Interaction Conference* (9-12 September 2014) *(HCI '04)*, Daniel Fitton, Matthew Horton, Janet C. Read, and Gavin Sim (Eds.). British Computer Society, Oxford. https://www.scienceopen.com/document?vid=d857ccd9-4660-4627-9d71-ee2db9d1d28a

[19] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. 2003. A Unifying Reference Framework for multi-target user interfaces. *Interacting with Computers* 15, 3 (6 2003), 289–308. https://doi.org/10.1016/S0953-5438(03)00010-9 arXiv:http://oup.prod.sis.lan/iwc/article-pdf/15/3/289/7800242/iwc15-0289.pdf

[20] David Chamberlain, Angel Díaz, Dan Gisolfi, Ravi B. Konuru, John M. Lucassen, Julie MacNaught, Stéphane H. Maes, Roland Merrick, David Mundel, T. V. Raman, Shankar Ramaswamy, Thomas Schaeck, Richard Thompson, and Charles Wiecha. 2002. WSXL: A Web Services Language for Integrating End-User Experience. In *Proceedings of the Fourth International Conference on Computer-Aided Design of User Interfaces, Computer-Aided Design of User Interfaces III* (May 15-17, 2002), Christophe Kolski and Jean Vanderdonckt (Eds.). Kluwer, Dordrecht, 35–50. https:

//doi.org/10.1007/978-94-010-0421-3_3

[21] Andreas Kuntner Hilda Tellioglu Christopher Mayer, Martin Morandell. 2013. A Comparison of User Description Languages Concerning Adaptability based on User Preferences. In *Proceedings of the 12th European Association for the Advancement of Assistive Technology in Europe Conference–Assistive Technology: From Research to Practice* (September 2013, 19-22) *(AAATE '13)*, Gert Jan Gelderblom Alan Newell Niels-Erik Mathiassen Pedro Encarnação, Luís Azevedo (Ed.). 1310–1315. https://doi.org/10.3233/978-1-61499-304-9-1310 Vol. 33.

[22] M. Cohn. 2004. *User Stories Applied for Agile Software Development* (1 ed.). Pearson Education Inc., Boston, MA, USA. Accessible at http://athena.ecs.csus.edu/~buckley/CSc191/User-Stories-Applied-Mike-Cohn.pdf.

[23] Benoît Collignon, Jean Vanderdonckt, and Gaëlle Calvary. 2008. Model-Driven Engineering of Multi-target Plastic User Interfaces. In *Proceedings of the Fourth IEEE International Conference on Autonomic and Autonomous Systems (ICAS'08)*. 7–14. https://doi.org/10.1109/ICAS.2008.37

[24] Simon Crowle and Linda Hole. 2003. ISML: An Interface Specification Meta-language. In *Proceedings of the 10th International Workshop on Design, Specification, and Verification of Interactive Systems* (June 11-13, 2003) *(DSV-IS 2003)*, Joaquim A. Jorge, Nuno Jardim Nunes, and João Falcão e Cunha (Eds.). Springer, Berlin, Heidelberg, 362–376. https://doi.org/10.1007/978-3-540-39929-2_25

[25] Lirisnei Gomes de Sousa and Jair C Leite. 2003. XICL: A Language for the User's Interfaces Development and Its Components. In *Proceedings of the Latin American Conference on Human-computer Interaction* (August 17-20, 2003) *(CLIHC '03)*. ACM, New York, NY, USA, 191–200. https://doi.org/10.1145/944519.944539

[26] Lirisnei Gomes de Sousa and Jair C. Leite. 2005. XICL — An Extensible Mark-up Language for Developing User Interface and Components. In *Proceedings of the Fifth International Conference on Computer-Aided Design of User Interfaces, Computer-Aided Design of User Interfaces IV* (13–16 January 2004) *(CADUI '04)*, Robert J.K. Jacob, Quentin Limbourg, and Jean Vanderdonckt (Eds.). Springer Netherlands, Dordrecht, 247–258. https://doi.org//10.1007/1-4020-3304-4_20

[27] Antonio Delgado, Antonio Estepa, J.A. Troyano, and Rafael M. Estepa. 2016. Reusing UI elements with Model-Based User Interface Development. *International Journal of Human-Computer Studies* 86 (2016), 48–62. https://doi.org/10.1016/j.ijhcs.2015.09.003

[28] Anne-Marie Déry-Pinna, Audrey Occello, and Michel Riveill. 2012. Towards Conflict Management in User Interface Composition Driven by Business Needs. In *Proceedings of 4th International Conference on Human-Centered Software Engineering* (October 29-31, 2012) *(HCSE '12)*, Marco Winckler, Peter Forbrig, and Regina Bernhaupt (Eds.). Springer, Berlin, Heidelberg, 233–250. https://doi.org/10.1007/978-3-642-34347-6_14

[29] Alain Derycke, Vincent Chevrin, and José Rouillard. 2005. Some issues for the Modelling of Interactive E-Services from the Customer Multi-Channel Interaction Perspectives. In *Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service*. IEEE Computer Society, Los Alamitos, CA, USA, 256–259. https://doi.org/10.1109/EEE.2005.123

[30] Hassan Djirdeh, Nate Murray, and Ari Lerner. 2018. *Fullstack Vue: The Complete Guide to Vue.js*. CreateSpace Independent Publishing Platform, FullStack.io. https://www.newline.co/vue

[31] Bruno Dumas, Denis Lalanne, and Rolf Ingold. 2010. Description languages for multimodal interaction: a set of guidelines and its illustration with SMUIML. *Journal on Multimodal User Interfaces* 3, 3 (01 Apr 2010), 237–247. https://doi.org/10.1007/s12193-010-0043-3

[32] Bruno Dumas, Beat Signer, and Denis Lalanne. 2014. A graphical editor for the SMUIML multimodal user interaction description language. *Science of Computer Programming* 86 (2014), 30 – 42. https://doi.org/10.1016/j.scico.2013.04.003 Special issue on Software Support for User Interface Description Languages (UIDL 2011).

[33] Bonnie Eisenman. 2016. *Learning React Native: Building Native Mobile Apps with JavaScript*. O'Reilly Media, Sebastopol, CA, USA. https://www.amazon.com/Learning-React-Native-Building-JavaScript-ebook/dp/B076PWRYSS

[34] Jacob Eisenstein, Jean Vanderdonckt, Jean Vanderdonckt, and Angel Puerta. 2000. Adapting to mobile contexts with user-interface modeling. In *Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications* (December 7-8, 2000). Institute of Electrical and Electronics Engineers, Los Alamitos, CA, USA, 83–92. https://doi.org/10.1109/MCSA.2000.895384

[35] Jacob Eisenstein, Jean Vanderdonckt, Jean Vanderdonckt, and Angel Puerta. 2001. Applying Model-based Techniques to the Development of UIs for Mobile Computers. In *Proceedings of the 6th ACM International Conference on Intelligent User Interfaces* (January 14-17, 2001) *(IUI '01)*. ACM, New York, NY, USA, 69–76. https://doi.org/10.1145/359784.360122

[36] Pablo Figueroa. 2009. InTml: A Case Study on Virtual Reality Development. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications* (October 25-29, 2009) *(OOPSLA '09)*. ACM, New York, NY, USA, 745–746. https://doi.org/10.1145/1639950.1639994

[37] Pablo Figueroa, Mark Green, and H. James Hoover. 2002. InTml: A Description Language for VR Applications. In *Proceedings of the Seventh International Conference on 3D Web Technology* (February 24 - 28, 2002) *(Web3D '02)*. ACM, New York, NY, USA, 53–58. https://doi.org/10.1145/504502.504511

[38] José Manuel Cantera Fonseca. 2010. Model-Based UI XG Final Report. (May 2010). https://www.w3.org/2005/Incubator/model-based-ui/XGR-mbui-20100504/ Contributors: Juan M. Gonzalez Calleros, Gerrit Meixner, Fabio Paterno, Jaroslav Pullmann, Dave Raggett, Daniel Schwabe, Jean Vanderdonckt.

[39] Borja Gamecho, Raúl Miñón, and Julio Abascal. 2013. Design Issues in Accessible User Interface Generation for Ubiquitous Services through Egoki. In *Proceedings of the 12th European Association for the Advancement of Assistive Technology in Europe Conference–Assistive Technology: From Research to Practice* (September 2013, 19-22) *(AAATE '13)*, Gert Jan Gelderblom Alan Newell Niels-Erik Mathiassen Pedro Encarnação, Luís Azevedo (Ed.). IOS Press, Amsterdam, 1304–1309. https://doi.org/10.3233/978-1-61499-304-9-1304 Vol. 33.

[40] Lamia Gaouar, Abdelkrim Benamar, Olivier Le Goaer, and Frédérique Biennier. 2018. HCIDL: Human-computer interface description language for multi-target, multimodal, plastic user interfaces. *Future Computing and Informatics Journal* 3, 1 (2018), 110–130. https://doi.org/10.1016/j.fcij.2018.02.001

[41] Steffen Goebel, Sven Buchholz, Thomas Ziegert, and Alexander Schill. 2002. Software Architecture for the Adaptation of Dialogs and Contents to Different Devices. In *Proceedings of International Conference on Information Networking: Wireless Communications Technologies and Network Applications* (January 30–February 1, 2002) *(ICOIN '02)*, Ilyoung Chong (Ed.). Springer, Berlin, Heidelberg, 42–51. https://doi.org/10.1007/3-540-45801-8_5

[42] Josefina Guerrero-Garcia, Juan Manuel Gonzalez-Calleros, and Jean Vanderdonckt. 2011. State of the Art of User Interface Description Languages. (February 2011). https://itea3.org/project/workpackage/document/download/468/08026-UsiXML-WP-1-D11v2-StateoftheArtofUIDL Deliverable D1.1, ITEA3 UsiXML Project.

[43] Josefina Guerrero-Garcia, Juan-Manuel Gonzalez-Calleros, Jean Vanderdonckt, and Jaime Munoz-Arteaga. 2009. A Theoretical Survey of User Interface Description Languages: Preliminary Results. In *Proceedings of IEEE International Latin American Web Congress (LA-Web '09)*. IEEE Press, Piscataway, New Jersey, USA, 36–43. https://doi.org/10.1109/LA-WEB.2009.40

[44] James Helms, Robbie Schaefer, Kris Luyten, Jo Vermeulen, Marc Abrams, Adrien Coyette, and Jean Vanderdonckt. 2009. *Human-Centered Engineering Of Interactive Systems With The User Interface Markup Language*. Springer, London, 139–171. https://doi.org/10.1007/978-1-84800-907-3_7

[45] Simon Holmes. 2015. *Getting MEAN with Mongo, Express,Angular and Node*. Manning Publications, Shelter Island, New York, United States.

[46] ECMA International. 2017. Standard ECMA-404 – The JSON Data Interchange Syntax. https://www.ecma-international.org/publications/standards/Ecma-404.htm

[47] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. 2017. Developer Mistakes in Writing Android Manifests: An Empirical Study of Configuration Errors. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE Press, 25–36. https://doi.org/10.1109/MSR.2017.41

[48] Paul D. Johnson and Jinesh Parekh. 2003. *Multiple Device Markup Language– a Rule Approach*. Master's thesis. DePaul University, Chicago, IL 60614, United States. http://se.cs.depaul.edu/ise/zoom/projects/ui/FINAL.pdf SE690 Research Seminar Final Report.

[49] William Jones. 2011. XooML: XML in Support of Many Tools Working on a Single Organization of Personal Information. In *Proceedings of the International iConference* (February 8-11, 2011) *(iConference '11)*. ACM, New York, NY, USA, 478–488. https://doi.org/10.1145/1940761.1940827

[50] William Jones and Kenneth M. Anderson. 2011. Many Views, Many Modes, Many Tools ... One Structure: Towards a Non-disruptive Integration of Personal Information. In *Proceedings of the 22nd ACM International Conference on Hypertext and Hypermedia* (June 6-9, 2011) *(HT '11)*. ACM, New York, NY, USA, 113–122. https://doi.org/10.1145/1995966.1995984

[51] Mlađan Jovanović, Dušan STARČEVIĆ, and Zoran JOVANOVIĆ. 2013. Languages for Model-Driven Development of User Interfaces: Review of the State of the Art. *Yugoslav Journal of Operations Research* 23, 3 (2013), 327–341. https://doi.org/10.2298/YJOR121101007J

[52] Picot-Coupey Karine, Huré Elodie, and Piveteau Lauren. 2016. Moving Towards an Omni-Channel Strategy: Process and Challenges. In *Celebrating America's Pastimes: Baseball, Hot Dogs, Apple Pie and Marketing?*, Kacy K. Kim (Ed.). Springer International Publishing, Cham, 523–523.

[53] Kouichi Katsurada, Yusaku Nakamura, Hirobumi Yamada, and Tsuneo Nitta. 2003. XISL: A Language for Describing Multimodal Interaction Scenarios. In *Proceedings of the 5th International Conference on Multimodal Interfaces* (November 5-7, 2003) *(ICMI '03)*. ACM, New York, NY, USA, 281–284. https://doi.org/10.1145/958432.958483

[54] Stefan Kost. 2004. Dynamically generated multi-modal application interfaces. In *Developing user interfaces with XML: Advances on User Interface Description Languages, Proceedings of Satellite Workshop of ACM International Conference on Advanced Visual Interfaces* (May 2004) *(UIDL '04)*, Kris Luyten, Marc Abrams, Jean Vanderdonckt, and Quentin Limbourg (Eds.). 231–244. https://www.semanticscholar.org/paper/Dynamically-generated-multi-modal-application-paper-Kost-Dresden/a64bb8acdb0647867dd117f35ee6b6d16460644a

[55] Stefan Kost. 2006. *Dynamically generated multi-modal application interfaces.* Ph.D. Dissertation. Technical University Dresden, Leipzig. http://gitk.sourceforge.net/docs/PhDT-StefanKost.pdf

[56] Yeonhee Lee and Hongsik Cheon. 2019. A Study on the Factors Affecting the User Intention of Omnichannel Shopping Based on Information Technology. In *Proceedings of the 2019 5th International Conference on E-Business and Applications (ICEBA 2019)*. Association for Computing Machinery, New York, NY, USA, 20–24. https://doi.org/10.1145/3317614.3317623

[57] Jair C. Leite. 2007. A Model-Based Approach to Develop Interactive System Using IMML. In *Task Models and Diagrams for Users Interface Design*, Karin Coninx, Kris Luyten, and Kevin A. Schneider (Eds.). Springer, Berlin, Heidelberg, 68–81. https://doi.org/10.1007/978-3-540-70816-2_6

[58] Jair C. Leite and Lirisnei de Sousa. 2008. The IMML VDE: models, languages and tools to develop interactive systems. *nternational Journal of Web Engineering and Technology* 4, 2 (2008), 183–206. https://doi.org/10.1504/IJWET.2008.018097

[59] Sophie Lepreux, Anas Hariri, José Rouillard, Dimitri Tabary, Jean-Claude Tarby, and Christophe Kolski. 2007. Towards Multimodal User Interfaces Composition Based on UsiXML and MBD Principles. In *Proceedings of 12th International Conference on Human-Computer Interaction. HCI Intelligent Multimodal Interaction Environments* (July 22-27, 2007) *(HCI International '07)*, Julie A. Jacko (Ed.). Springer, Berlin, Heidelberg, 134–143. https://doi.org/10.1007/978-3-540-73110-8_15

[60] Sophie Lepreux and Jean Vanderdonckt. 2007. Towards A Support of User Interface Design By Composition Rules. In *Computer-Aided Design of User Interfaces V: Proceedings of the Sixth International Conference on Computer-Aided Design of User Interfaces* (June 6-8, 2006) *(CADUI '06)*, Gaëlle Calvary, Costin Pribeanu, Giuseppe Santucci, and Jean Vanderdonckt (Eds.). Springer Netherlands, Dordrecht, 231–244. https://doi.org/10.1007/978-1-4020-5820-2_19

[61] Sophie Lepreux, Jean Vanderdonckt, and Benjamin Michotte. 2007. Visual Design of User Interfaces by (De)composition. In *Proceedings of 13th International Workshop on Design, Specification, and Verification of Interactive Systems* (July 26-28, 2006) *(DSV-IS '06)*, Gavin Doherty and Ann Blandford (Eds.). Springer, Berlin, Heidelberg, 157–170. https://doi.org/10.1007/978-3-540-69554-7_13

[62] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Murielle Florins. 2004. USIXML: A User Interface Description Language Supporting Multiple Levels of Independence. In *Proceedings of Workshops in connection with the 4th International Conference on Web Engineering (ICWE '04). Engineering Advanced Web Applications* (28-30 July, 2004) *(DIWE '04)*, Maristella Matera and Sara Comai (Eds.). Rinton Press, 325–338.

[63] Kris Luyten, Marc Abrams, Jean Vanderdonckt, and Quentin Limbourg. 2004. Developing user interfaces with XML: Advances on User Interface Description Languages. *Proceedings of Satellite Workshop of ACM International Conference on Advanced Visual Interfaces* (2004). https://dial.uclouvain.be/downloader/downloader.php?pid=boreal:168618&datastream=PDF_01

[64] Kris Luyten and Karin Coninx. 2001. An XML-Based Runtime User Interface Description Language for Mobile Computing Devices. In *Proceedings of 8th International Workshop on Design, Specification, and Verification of Interactive Systems* (June 13–15, 2001) *(DSV-IS '01)*, Chris Johnson (Ed.). Springer, Berlin, Heidelberg, 1–15. https://doi.org/10.1007/3-540-45522-1_1

[65] Gerrit Meixner, Gaëlle Calvary, and Joëlle Coutaz. 2014. Introduction to Model-Based User Interfaces - W3C Working Group Note. https://www.w3.org/TR/mbui-intro/

[66] Gerrit Meixner, Fabio Paternò, and Jean Vanderdonckt. 2011. Past, Present, and Future of Model-Based User Interface Development. *i-com Zeitschrift für interaktive und kooperative Medien* 10, 3 (2011), 2–11. https://doi.org/10.1524/icom.2011.0026

[67] Gerrit Meixner, Marc Seissler, and Kai Breiner. 2011. *Model-Driven Useware Engineering*. Studies in Computational Intelligence, Vol. 340. Springer, Berlin, Heidelberg, 1–26. https://doi.org/10.1007/978-3-642-14562-9_1

[68] Gerrit Meixner, Marc Seissler, and M. Nahler. 2009. Udit: A Graphical Editor For Task Models. In *4th International Workshop on Model Driven Development of Advanced User Interfaces. International Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI-2009), February 8, Sanibel Island, Florida, United States (CEUR Workshop Proceedings)*, Gerrit Meixner, Daniel Görlich, K. Breiner, H. Hußmann, A. Pleuß, S. Sauer, and J. Van den Bergh (Eds.), Vol. 439. CEUR Workshop Proceedings (Online).

[69] Gerrit Meixner, Marc Seissler, and Marius Orfgen. 2012. Specification and Application of a Taxonomy for Task Models in Model-Based User Interface Development Environments. *International Journal on Advances in Intelligent Systems* 4, 3+4 (4 2012), 388–398.

[70] Jérémie Melchior, Jean Vanderdonckt, and Peter Van Roy. 2011. A Model-based Approach for Distributed User Interfaces. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (June 13-16, 2011) *(EICS '11)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/1996461.1996488

[71] Kyle Mew. 2015. *Learning Material Design.* Packt Publishing, Birmingham, United Kingdom. Accessible at https://www.packtpub.com/web-development/learning-material-design.

[72] Benjamin Michotte and Jean Vanderdonckt. 2008. GrafiXML, a Multi-target User Interface Builder Based on UsiXML. In *Fourth International Conference on Autonomic and Autonomous Systems, 16-21 March 2008, Gosier, Guadeloupe (ICAS '08)*. IEEE Computer Society, 15–22. https://doi.org/10.1109/ICAS.2008.29

[73] Nikola Mitrovic, Carlos Bobed, and Eduardo Mena. 2016. A Review of User Interface Description Languages for Mobile Applications. In *Proceedings of the Tenth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM '16)*. International Academy, Research, and Industry Association (IARIA), Wilmington, DE, USA, 96–101. https://www.thinkmind.org/index.php?view=article&articleid=ubicomm_2016_6_10_10063

[74] Andreas Mladenow, Antoaneta Mollova, and Christine Strauss. 2018. Mobile Technology Contributing to Omni-Channel Retail. In *Proceedings of the 16th International Conference on Advances in Mobile Computing and Multimedia (MoMM2018)*. Association for Computing Machinery, New York, NY, USA, 92–101. https://doi.org/10.1145/3282353.3282371

[75] Pedro J. Molina. 2018. Quid: A web-based DSL for defining User Interfaces applied to Web Components. In *Actas de las XXIII Jornadas de Ingeniería del Software y Bases de Datos. , septiembre de 2018* (September 17-19, 2018) *(JISBD '18)*, F. Sánchez-Figueroa (Ed.). Biblioteca Digital de SISTEDES, Valencia, Spain. https://doi.org/11705/JISBD/2018/067

[76] Pedro J. Molina. 2019. Quid: Prototyping Web Components on the Web. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (June 18-21, 2019) *(EICS '19)*. ACM, New York, NY, USA, Article 3, 5 pages. https://doi.org/10.1145/3319499.3330294

[77] Pedro J. Molina, Santiago Meliá, and Oscar Pastor. 2002. User Interface Conceptual Patterns. In *Interactive Systems:Design, Specification, and Verification*, Peter Forbrig, Quentin Limbourg, Jean Vanderdonckt, and Bodo Urban (Eds.). Springer, Berlin, Heidelberg, 159–172.

[78] Francisco Montero, Víctor López-Jaquero, Jean Vanderdonckt, Pascual González, María Lozano, and Quentin Limbourg. 2006. Solving the Mapping Problem in User Interface Design by Seamless Integration in IdealXML. In *Proceedings of the 12th International Workshop on Design, Specification, and Verification of Interactive Systems* (July 13-15, 2006) *(DSV-IS '06)*, Stephen W. Gilroy and Michael D. Harrison (Eds.). Springer, Berlin, Heidelberg, 161–172. https://doi.org/10.1007/11752707_14

[79] Jason Morris. 2011. *Android User Interface Development: Beginner's Guide*. Packt Publishing, Birmingham, United Kingdom. Accessible at https://www.packtpub.com/eu/application-development/android-user-interface-development-beginners-guide.

[80] Vivian Genaro Motti and Jean Vanderdonckt. 2013. A computational framework for context-aware adaptation of user interfaces. In *Proceedings of the IEEE 7th International Conference on Research Challenges in Information Science* (May 29-31, 2013) *(RCIS '13)*. 1–12. https://doi.org/10.1109/RCIS.2013.6577709

[81] Nate Murray, Felipe Coury, Ari Lerner, and Carlos Taborda. 2019. *ng-book, The Complete Book on Angular*. CreateSpace Independent Publishing Platform, FullStack.io. https://www.newline.co/ng-book/2/

[82] Brad Myers, Scott E. Hudson, and Randy Pausch. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.* 7, 1 (March 2000), 3–28. https://doi.org/10.1145/344949.344959

[83] Brad A. Myers and Mary Beth Rosson. 1992. Survey on User Interface Programming. In *Proceedings of the ACM International Conference on Human Factors in Computing Systems* (May 3-7, 1992) *(CHI '92)*. ACM, New York, NY, USA, 195–202. https://doi.org/10.1145/142750.142789

[84] Adam Nathan. 2016. *Building Windows 10 Applications with XAML and C Unleashed* (2 ed.). Sams Publishing, Indianapolis, Indiana, USA. Accessible at https://www.hpb.com/products/building-windows-10-applications-with-xaml-and-c-unleashed-9780672337581.

[85] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. 2005. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In *Proceedings of the ACM International Workshop on Mining Software Repositories (MSR '05)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/1083142.1083143

[86] Michael Nebeling, Michael Grossniklaus, Stefania Leone, and Moira C. Norrie. 2012. XCML: providing context-aware language extensions for the specification of multi-device web applications. *World Wide Web* 15, 4 (01 Jul 2012), 447–481. https://doi.org/10.1007/s11280-011-0152-2

[87] Thanh-Diane Nguyen, Jean Vanderdonckt, and Ahmed Seffah. 2016. UIPLML: Pattern-based Engineering of User Interfaces of Multi-platform Systems. In *Proceedings of the IEEE Tenth International Conference on Research Challenges in Information Science* (June 1-3, 2016) *(RCIS '16)*. 1–12. https://doi.org/10.1109/RCIS.2016.7549348

[88] Hartmut Obendorf and Matthias Finck. 2008. Scenario-Based Usability Engineering Techniques in Agile Development Processes. In *CHI '08 Extended Abstracts on Human Factors in Computing Systems (CHI EA '08)*. Association for Computing Machinery, New York, NY, USA, 2159–2166. https://doi.org/10.1145/1358628.1358649

[89] Audrey Occello, Cédric Joffroy, Anne-Marie Dery-Pinna, Philippe Renevier-Gonin, and Michel Riveill. 2011. Metamodeling functional and interactive parts of systems for composition considerations. *Journal of Computational Methods in Sciences and Engineering* 11, 1 (2011), 103–113. https://doi.org/10.3233/JCM-2011-0381

[90] Hector Olmedo, David Escudero, and Valentín Cardeñoso. 2015. Multimodal interaction with virtual worlds XMMVR: eXtensible language for MultiModal interaction with virtual reality worlds. *Journal on Multimodal User Interfaces* 9, 3 (01 Sep 2015), 153–172. https://doi.org/10.1007/s12193-015-0176-5

[91] Object Management Group (OMG). 2008. Software Systems Process Engineering Meta-Model Specification. https://www.omg.org/spec/SPEM/2.0/PDF

[92] Arum Park, Jae Yoon Han, and Kyoung Jun Lee. 2017. IoT-Based Omni Channel Service for Smart Exhibition and Value of Data. In *Proceedings of the International Conference on Electronic Commerce (ICEC '17)*. Association for Computing Machinery, New York, NY, USA, Article 10, 10 pages. https://doi.org/10.1145/3154943.3154966

[93] Fabio Paterno', Carmen Santoro, and Lucio Davide Spano. 2009. MARIA: A Universal, Declarative, Multiple Abstraction-level Language for Service-oriented Applications in Ubiquitous Environments. *ACM Transactions on Computer-Human Interaction* 16, 4, Article 19 (Nov. 2009), 30 pages. https://doi.org/10.1145/1614390.1614394

[94] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. 2011. Engineering the authoring of usable service front ends. *Journal of Systems and Software* 84, 10 (2011), 1806–1822. https://doi.org/10.1016/j.jss.2011.05.025

[95] Petar Petrovski, Anna Primpeli, Robert Meusel, and Christian Bizer. 2017. The WDC Gold Standards for Product Feature Extraction and Product Matching. In *E-Commerce and Web Technologies*, Derek Bridge and Heiner Stuckenschmidt (Eds.). Springer International Publishing, Cham, 73–86.

[96] Jaroslav Pullmann. 2014. Model-Based User Interface Charter - Glossary. W3C Working Group Note. https://www.w3.org/TR/mbui-glossary/

[97] Juan C. Quiroz, Sergiu M. Dascalu, and Sushil J. Louis. 2007. Human Guided Evolution of XUL User Interfaces. In *CHI '07 Extended Abstracts on Human Factors in Computing Systems (CHI EA '07)*. Association for Computing Machinery, New York, NY, USA, 2621–2626. https://doi.org/10.1145/1240866.1241052

[98] David Raneburger, Roman Popp, and Jean Vanderdonckt. 2012. An Automated Layout Approach for Model-driven WIMP-UI Generation. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (June 25-26, 2012) *(EICS '12)*. ACM, New York, NY, USA, 91–100. https://doi.org/10.1145/2305484.2305501

[99] Ray Rischpater. 2014. *Application Development with Qt Creator* (2 ed.). Packt Publishing, Birmingham, United Kingdom. Accessible at https://www.packtpub.com/application-development/application-development-qt-creator-second-edition.

[100] José Rouillard. 2003. Plastic ML and its toolkit. In *Proceedings of the 10th International Conference on Human Computer Interaction* (June 22-27, 2003) *(HCI International '03)*, Constantine Stephanidis (Ed.). Lawrence Erlbaum Associates, Inc, 612–616. http://jose.rouillard.free.fr/PlasticML/

[101] Jenny Ruiz, Estefanía Serral, and Monique Snoeck. 2018. Evaluating user interface generation approaches: model-based versus model-driven development. *Software & Systems Modeling* (17 October 2018), 2753—-2776. https://doi.org/10.1007/s10270-018-0698-x

[102] Ugo Braga Sangiorgi, François Beuvens, and Jean Vanderdonckt. 2012. User Interface Design by Collaborative Sketching. In *Proceedings of the ACM International Conference on Designing Interactive Systems* (June 11-15, 2012) *(DIS '12)*. ACM, New York, NY, USA, 378–387. https://doi.org/10.1145/2317956.2318013

[103] Robbie Schaefer, Steffen Bleul, and Wolfgang Mueller. 2007. Dialog Modeling for Multiple Devices and Multiple Interaction Modalities. In *Proceedings of 5th International Workshop on Task Models and Diagrams for Users Interface Design* (October 23-24, 2006) *(TAMODIA '06)*, Karin Coninx, Kris Luyten, and Kevin A. Schneider (Eds.). Springer, Berlin, Heidelberg, 39–53. https://doi.org/10.1007/978-3-540-70816-2_4

[104] Orit Shaer, Robert J. K. Jacob, Mark Green, and Kris Luyten. 2008. User Interface Description Languages for Next Generation User Interfaces. In *Proceedings of the ACM International Conference on Human Factors in Computing Systems, Extended Abstracts* (April 5-10, 2008) *(CHI EA '08)*. ACM, New York, NY, USA, 3949–3952. https://doi.org/10.1145/1358628.1358964

[105] Fanjuan Shi. 2017. Omni-Channel Retailing: Knowledge, Challenges, and Opportunities for Future Research. In *Marketing at the Confluence between Entertainment and Analytics*, Patricia Rossi (Ed.). Springer International Publishing, Cham, 91–102.

[106] João Carlos Silva, João Saraiva, and José Creissac Campos. 2009. A Generic Library for GUI Reasoning and Testing. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC '09)*. Association for Computing Machinery, New York, NY, USA, 121–128. https://doi.org/10.1145/1529282.1529307

[107] Nathalie Souchon and Jean Vanderdonckt. 2003. A Review of XML-compliant User Interface Description Languages. In *Proceedings of 10th International Workshop of Design, Specification, and Verification of Interactive Systems* (June 11-13, 2003) *(DSV-IS '03, Lecture Notes in Computer Science)*, Joaquim A. Jorge, Nuno Jardim Nunes, and João Falcão e Cunha (Eds.), Vol. 2844. Springer-Verlag, Berlin, Heidelberg, 377–391. https://doi.org/10.1007/978-3-540-39929-2_26

[108] Jordan Danford Dan Zajdband Vu Tran Stijn de Witt, Oskar Hane. 2018. *Progressive Web Apps with Preact*. Bleeding Edge Press, bleedingedgepress.com. https://www.oreilly.com/library/view/progressive-web-apps/9781939902535/

[109] Pedro A. Szekely and Brad A. Myers. 1988. A User Interface Toolkit Based on Graphical Objects and Constraints. In *Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications* (September 25-30, 1988) *(OOPSLA '88)*, Norman K. Meyrowitz (Ed.). ACM, 36–45. https://doi.org/10.1145/62083.62088

[110] Seika Tanaka, Hajime Iwata, Junko Shirogane, and Yoshiaki Fukazawa. 2019. Development Support of User Interfaces Adaptive to Use Environment. In *Proceedings of the 8th International Conference on Software and Computer Applications* (February 19-21, 2019) *(ICSCA '19)*. ACM, New York, NY, USA, 223–228. https://doi.org/10.1145/3316615.3316663

[111] T.J. Van Toll. 2014. *jQuery UI in Action* (1 ed.). Manning Publications, Shelter Island, New York, USA. Accessible at https://livebook.manning.com/book/jquery-ui-in-action/about-this-book/0.

[112] Jean Vanderdonckt. 2008. Model-Driven Engineering of User Interfaces: Promises, Successes, and Failures. In *Proceedings of the 5th International Romanian Conference on Human-Computer Interaction* (September 18-19, 2008) *(RoCHI '08)*, Sabin Buraga and Ion Juvina (Eds.). Matrix ROM, Bucharest, Romania, 1–10. http://hdl.handle.net/2078/118090

[113] Jean Vanderdonckt and Thanh-Diane Nguyen. 2019. MoCaDiX: Designing Cross-Device User Interfaces of an Information System Based on Its Class Diagram. *Proceedings of ACM Human-Computer Interaction* 3, EICS, Article 17 (June 2019), 40 pages. https://doi.org/10.1145/3331159

[114] Arun Veeramani, Kausik Venkatesan, and K Nalinadevi. 2014. Abstract Syntax Tree Based Unified Modeling Language to Object Oriented Code Conversion. In *Proceedings of the 2014 International Conference on Interdisciplinary Advances in Applied Computing (ICONIAAC '14)*. Association for Computing Machinery, New York, NY, USA, Article Article 25, 8 pages. https://doi.org/10.1145/2660859.2660934

[115] Marco Winckler, Jean Vanderdonckt, Adrian Stanciulescu, and Francisco Trindade. 2008. Cascading Dialog Modeling with UsiXML. In *Interactive Systems. Design, Specification, and Verification*, T. C. Nicholas Graham and Philippe Palanque (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 121–135.

[116] Edd Yerburgh. 2018. *Testing Vue.js Applications*. Manning Publications, Shelter Island, New York, United States. https://books.google.be/books?id=7-FAtAEACAAJ

[117] Vadim Zaytsev. 2012. BNF Was Here: What Have We Done about the Unnecessary Diversity of Notation for Syntactic Definitions. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12)*. Association for Computing Machinery, New York, NY, USA, 1910–1915. https://doi.org/10.1145/2245276.2232090

[118] Thomas Ziegert, Markus Lauff, and Lutz Heuser. 2004. Device Independent Web Applications – The Author Once – Display Everywhere Approach. In *Proceedings of 4th International Conference on Web Engineering* (July 26-30, 2004) *(ICWE '04)*, Nora Koch, Piero Fraternali, and Martin Wirsing (Eds.). Springer, Berlin, Heidelberg, 244–255. https://doi.org/10.1007/978-3-540-27834-4_31

[119] Gottfried Zimmermann, Gregg Vanderheiden, and Al Gilman. 2003. Universal Remote Console - Prototyping for the Alternate Interface Access Standard. In *Proceedings of 7th ERCIM International Workshop on User Interfaces for All – Universal Access Theoretical Perspectives, Practice, and Experience* (October 24–25, 2002) *(UI4All '02)*, Noëlle Carbonell and Constantine Stephanidis (Eds.). Springer, Berlin, Heidelberg, 524–531. https://doi.org/10.1007/3-540-36572-9_40

[120] Detlef Zuehlke and Nancy Thiels. 2008. Useware engineering: a methodology for the development of user-friendly interfaces. *Library Hi Tech* 26, 1 (March 2008), 126–140. https://doi.org/10.1108/07378830810857852

APPENDIX A. JSON DEFINITION OF A OPENUIDL COMPONENT

This appendix reproduces the concrete syntax of OpenUIDL components, expressed as a JSON (JavaScript Object Notation) file [46], a commonly used lightweight data-interchange format, that is expected to be relatively easy for both humans to read and write and for machines to parse and process, such as for code generation.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "component.json",
  "name": "Component UIDL",
  "type": "object",
  "required": ["name", "node"],
  "additionalProperties": false,
  "properties": {
    "$schema": {
      "type": "string",
      "format": "uri"
    },
    "name": {
      "type": "string",
      "default": "MyComponent"
    },
    "node": {
      "type": "object",
      "additionalProperties": false,
      "required": ["type", "content"],
      "properties": {
        "type": { "type": "string", "enum": ["element"] },
        "content": { "$ref": "#/definitions/elementNodeContent" }
      }
    },
    "outputOptions": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "componentClassName": { "type": "string", "pattern": "^[A
-Z]+[a-zA-Z0-9]*$" },
        "fileName": { "type": "string", "pattern": "^[a-zA-Z0-9-_
.]*$" },
        "styleFileName": { "type": "string", "pattern": "^[a-zA-Z
0-9-_.]*$" },
        "templateFileName": { "type": "string", "pattern": "^[a-
zA-Z0-9-_.]*$" },
        "moduleName": { "type": "string", "pattern": "^[a-zA-Z0-9
-_.]*$" },
        "folderPath": { "type": "array", "items": { "type": "
string", "pattern": "^[a-zA-Z0-9-_.]*$" } }
```

```
      }
    },
    "stateDefinitions":  { "$ref": "#/definitions/
  stateDefinitions" },
    "propDefinitions": { "$ref": "#/definitions/propDefinitions"
  }
},
"definitions": {
  "propDefinitions": {
    "type": "object",
    "patternProperties": {
      ".*": {
        "type": "object",
        "additionalProperties": false,
        "required": ["type"],
        "properties": {
          "type": {
            "type": "string",
            "enum": ["string", "boolean", "number", "array", "
  func", "object", "children"]
          },
          "defaultValue": {
            "oneOf": [
              { "type": "number" },
              { "type": "boolean" },
              { "type": "string" },
              { "type": "object" },
              {
                "type": "array",
                "items": {
                  "oneOf": [
                    { "type": "number" },
                    { "type": "string" },
                    { "type": "object" }
                  ]
                }
              }
            ]
          },
          "isRequired": {
            "type": "boolean"
          },
          "meta": {"type": "object"}
        }
      }
    }
  },
  "stateDefinitions": {
```

```
    "type": "object",
    "patternProperties": {
      ".*": {
        "type": "object",
        "additionalProperties": false,
        "required": ["type", "defaultValue"],
        "properties": {
          "type": {"type":"string", "enum": ["string", "boolean
", "number", "object", "func", "array", "router"]},
          "values": {
            "type": "array",
            "items": {
              "type": "object",
              "properties": {
                "value": {},
                "pageOptions": {
                  "type": "object",
                  "additionalProperties": false,
                  "properties": {
                    "componentName": { "type": "string", "
pattern": "^[A-Z]+[a-zA-Z0-9]*$" },
                    "navLink": { "type": "string", "pattern":
"^\/[a-zA-Z0-9-_]*$" },
                    "fileName": { "type": "string", "pattern":
"^[a-zA-Z0-9-_.]*$" }
                  }
                },
                "seo": {
                  "type": "object",
                  "additionalProperties": false,
                  "properties": {
                    "title": { "type": "string" },
                    "metaTags": {
                      "type": "array",
                      "items": {
                        "type": "object",
                        "patternProperties": {
                          ".*": {
                            "type": "string"
                          }
                        }
                      }
                    },
                    "assets": {
                      "type": "array",
                      "items": {
                        "type": "object",
                        "additionalProperties": false,
```

```
                        "required": ["type"],
                        "properties": {
                          "type": { "type": "string", "enum": [
"link", "script", "font", "icon", "style", "canonical"] },
                          "path": { "type": "string" },
                          "content": { "type": "string" },
                          "meta": { "type": "object" }
                        }
                      }
                    }
                  }
                }
              }
            },
            "defaultValue": {}
          }
        }
      }
    },
    "node": {
      "oneOf": [
        {
          "type": "string"
        },
        {
          "type":"object",
          "additionalProperties": false,
          "required": ["type", "content"],
          "properties": {
            "type": {
              "type":"string",
              "enum": ["static", "dynamic", "element", "repeat","
conditional","slot"]
            },
            "content": {
              "oneOf": [{
                "$ref": "#/definitions/staticValueContent"
              },{
                "$ref": "#/definitions/dynamicReferenceContent"
              },{
                "$ref": "#/definitions/repeatNodeContent"
              },{
                "$ref": "#/definitions/elementNodeContent"
              },{
                "$ref": "#/definitions/conditionalNodeContent"
              },{
                "$ref": "#/definitions/slotNodeContent"
```

```
            }]
          }
        },
        "if": {
          "properties": {
            "type": {
              "const": "static"
            }
          }
        },
        "then": {
          "properties": {
            "content": {
              "$ref": "#/definitions/staticValueContent"
            }
          }
        }
      }
    ]
  },
  "staticValueContent": {
    "oneOf": [
      { "type": "string" },
      { "type": "number" },
      { "type": "boolean" },
      { "type": "array"}
    ]
  },
  "dynamicReferenceContent": {
    "type": "object",
    "required": ["referenceType","id" ],
    "additionalProperties": false,
    "properties": {
      "referenceType" :{
        "type": "string",
        "enum":  ["prop", "state", "local", "attr"]
      },
      "id": { "type": "string" }
    }
  },
  "repeatNodeContent": {
    "type": "object",
    "required": [ "node", "dataSource" ],
    "properties": {
      "node":{ "$ref": "#/definitions/node"},
      "dataSource":{"$ref": "#/definitions/attributeValue"},
      "meta": {
        "type":"object",
```

```
        "properties": {
          "useIndex": {"type": "boolean"},
          "iteratorName": {"type": "string"},
          "dataSourceIdentifier": {"type": "string"}
        }
      }
    }
  },
  "attributeValue": {
    "type": "object",
    "properties": {
      "type": {
        "type": "string",
        "enum": ["static", "dynamic"]
      },
      "content": {
        "oneOf": [
          {"$ref":"#/definitions/dynamicReferenceContent"},
          {"$ref":"#/definitions/staticValueContent"}
        ]
      }
    }
  },
  "conditionalNodeContent": {
    "type":"object",
    "required": [ "node", "reference" ],
    "properties": {
      "node":{ "$ref": "#/definitions/node"},
      "reference":{
        "type": "object",
        "properties": {
          "type": {
            "type": "string",
            "enum": ["dynamic"]
          },
          "content": {"$ref":"#/definitions/
dynamicReferenceContent"}
        }
      },
      "value": {
        "oneOf": [
          { "type": "string" },
          { "type": "number" },
          { "type": "boolean" }
        ]
      },
      "condition": {"$ref":"#/definitions/conditionalExpression
"}
```

```
      }
    },
    "conditionalExpression":{
      "type":"object",
      "required": ["conditions"],
      "properties": {
        "conditions":  {
          "type":"array",
          "items": {
            "type":"object",
            "required": ["operation"],
            "properties": {
              "operation": {"type": "string"},
              "operand": {
                "oneOf": [
                  { "type": "string" },
                  { "type": "number" },
                  { "type": "boolean" }
                ]
              }
            }
          }
        },
        "matchingCriteria": { "type": "string" }
      }
    },
    "elementNodeContent": {
      "type": "object",
      "required": ["elementType"],
      "additionalProperties": false,
      "properties": {
        "additionalProperties": false,
        "elementType": {"type": "string" },
        "name": {"type": "string", "pattern": "^[a-zA-Z]+[a-zA-Z0
-9-_]*$"},
        "key": {"type": "string"},
        "dependency": {"$ref": "#/definitions/componentDependency
"},
        "style": {"$ref": "#/definitions/styleDefinitions"},
        "attrs": {
          "type": "object",
          "patternProperties": {
            ".*": { "oneOf": [
              { "$ref": "#/definitions/attributeValue" },
              { "type": "string" },
              { "type": "number" },
              { "type": "boolean" },
              { "type": "array"}
```

```
            ] }
          }
        },
        "events":{"$ref": "#/definitions/eventDefinitions"},
        "children": { "type": "array", "items": {"$ref": "#/
definitions/node"}}
      }
    },
    "componentDependency": {
      "type": "object",
      "required": ["type"],
      "properties": {
        "type": {"type": "string"},
        "path": {"type": "string"},
        "version": {"type": "string"},
        "meta": {
          "type":"object",
          "properties": {
            "namedImport": {"type": "boolean"},
             "originalName": {"type": "string"}
          }
        }
      }
    },
    "styleDefinitions":{
      "type": "object",
      "patternProperties": {
        ".*": { "oneOf": [
          { "$ref": "#/definitions/styleValue" },
          { "type": "string" },
          { "type": "number" },
          {
            "type": "object",
            "patternProperties": {
              ".*": {"$ref": "#/definitions/styleDefinitions"}
            }
          }
        ] }
      }
    },
    "styleValue": {
      "type": "object",
      "oneOf": [
        {"$ref":"#/definitions/attributeValue"},
        {"$ref":"#/definitions/nestedStyleDeclaration"}
      ]
    },
    "nestedStyleDeclaration": {
```

```
      "type": "object",
      "required": ["type", "content"],
      "properties":{
        "type": {
          "type": "string",
          "enum": ["nested-style"]
        },
        "content":{"$ref": "#/definitions/styleDefinitions"}
      }
    },
    "eventDefinitions":{
      "type": "object",
      "patternProperties": {
        ".*": {
          "type":"array",
          "items": {
            "type": "object",
            "required": ["type"],
            "properties": {
              "type": {"type": "string"},
              "modifies": {"type": "string"},
              "newState": {
                "oneOf": [
                  { "type": "string" },
                  { "type": "number" },
                  { "type": "boolean" }
                ]
              },
              "calls": {"type": "string"},
              "args": {
                "type": "array",
                "items": {
                  "oneOf": [
                    { "type": "string" },
                    { "type": "number" },
                    { "type": "boolean" }
                  ]
                }
              }
            }
          }
        }
      }
    },
    "slotNodeContent": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
```

```
        "fallback": { "$ref": "#/definitions/node" },
        "name": { "type": "string" }
      }
    }
  }
}
```

APPENDIX B. JSON DEFINITION OF A OPENUIDL PROJECT

This appendix reproduces the concrete syntax of a OpenUIDL project, expressed as a JSON (JavaScript Object Notation) file [46].

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "title": "Project UIDL",
  "required": [
    "$schema",
    "name",
    "root",
    "globals"
  ],
  "additionalProperties": false,
  "properties": {
    "$schema": {
      "type": "string",
      "format": "uri"
    },
    "name": {
      "type": "string",
%      "default": "My-Teleport-Project"
      "default": "My-Project"
    },
    "globals": {
      "type": "object",
      "additionalProperties": false,
      "required": ["settings", "assets", "meta"],
      "properties": {
        "settings": {
          "type": "object",
          "additionalProperties": false,
          "properties": {
            "language": { "type": "string" },
            "title": { "type": "string" }
          }
        },
        "meta": {
          "type": "array",
          "items": {
            "type": "object"
          }
        },
        "manifest": {
          "type": "object",
          "additionalProperties": false,
```

```
          "properties": {
            "short_name": { "type": "string" },
            "name": { "type": "string" },
            "icons": {
              "type": "array",
              "items": {
                "type": "object",
                "additionalProperties": false,
                "required": ["src", "type", "sizes"],
                "properties": {
                  "src": { "type": "string" },
                  "type": { "type": "string" },
                  "sizes": { "type": "string" }
                }
              }
            },
            "start_url": { "type": "string" },
            "background_color": { "type": "string" },
            "display": { "type": "string" },
            "orientation": { "type": "string" },
            "scope": { "type": "string" },
            "theme_color": { "type": "string" }
          }
        },
        "assets": {
          "type": "array",
          "items": {
            "type": "object",
            "additionalProperties": false,
            "required": ["type"],
            "properties": {
              "type": { "type": "string", "enum": ["link", "
    script", "font", "icon", "style", "canonical"] },
              "path": { "type": "string" },
              "content": { "type": "string" },
              "options": { "type": "object" }
            }
          }
        },
        "variables": {
          "type": "object"
        }
      }
    },
    "root": {
      "$ref": "component.json"
    },
    "components": {
```

```
      "type": "object",
      "patternProperties": {
        ".*": {
          "$ref": "component.json"
        }
      }
    }
  }
}
```

APPENDIX C. TYPESCRIPT INTERFACE DEFINITION OF OPENUIDL

This appendix reproduces the OpenUIDL syntax in terms of TypeScript interface, a structure that defines the contract in the target application. It defines the syntax for classes to follow. Classes that are derived from an interface must follow the structure provided by their interface.

```
export interface ProjectUIDL {
  $schema?: string
  name: string
  globals: UIDLGlobalProjectValues
  root: ComponentUIDL
  components?: Record<string, ComponentUIDL>
}

export interface UIDLGlobalProjectValues {
  settings: {
    title: string
    language: string
  }
  meta: Array<Record<string, string>>
  assets: UIDLGlobalAsset[]
  manifest?: WebManifest
  variables?: Record<string, string>
}

export interface UIDLGlobalAsset {
  type: 'script' | 'style' | 'font' | 'canonical' | 'icon'
  path?: string
  content?: string
  options?: {
    async?: boolean
    defer?: boolean
    target?: string
    iconType?: string
    iconSizes?: string
  }
}

export interface ComponentUIDL {
  $schema?: string
  name: string
```

```
  node: UIDLElementNode
  propDefinitions?: Record<string, UIDLPropDefinition>
  stateDefinitions?: Record<string, UIDLStateDefinition>
  outputOptions?: UIDLComponentOutputOptions
  seo?: UIDLComponentSEO
}

export interface UIDLComponentOutputOptions {
  componentClassName?: string // needs to be a valid class name
  fileName?: string // needs to be a valid file name
  styleFileName?: string
  templateFileName?: string
  moduleName?: string
  folderPath?: string[]
}

export interface UIDLComponentSEO {
  title?: string
  metaTags?: UIDLMetaTag[]
  assets?: UIDLGlobalAsset[]
}

export type UIDLMetaTag = Record<string, string>

export interface UIDLPropDefinition {
  type: string
  defaultValue?: string | number | boolean | unknown[] | object | (() => void)
  isRequired?: boolean
}

export interface UIDLStateDefinition {
  type: string
  defaultValue: string | number | boolean | unknown[] | object | (() => void)
  values?: UIDLStateValueDetails[]
}

export interface UIDLStateValueDetails {
  value: string | number | boolean
  pageOptions?: UIDLPageOptions // Used when the StateDefinition is used as the
    router
  seo?: UIDLComponentSEO
}

export interface UIDLPageOptions {
  componentName?: string
  navLink?: string
  fileName?: string
}

export type ReferenceType = 'prop' | 'state' | 'local' | 'attr' | 'children'

export interface UIDLDynamicReference {
```

```
  type: 'dynamic'
  content: {
    referenceType: ReferenceType
    id: string
  }
}

export interface UIDLStaticValue {
  type: 'static'
  content: string | number | boolean | unknown[] // unknown[] for data sources
}

export interface UIDLRawValue {
  type: 'raw'
  content: string
}

export interface UIDLSlotNode {
  type: 'slot'
  content: {
    name?: string
    fallback?: UIDLElementNode | UIDLStaticValue | UIDLDynamicReference
  }
}

export interface UIDLNestedStyleDeclaration {
  type: 'nested-style'
  content: UIDLStyleDefinitions
}

export interface UIDLRepeatNode {
  type: 'repeat'
  content: UIDLRepeatContent
}

export interface UIDLRepeatContent {
  node: UIDLElementNode
  dataSource: UIDLAttributeValue
  meta?: UIDLRepeatMeta
}

export interface UIDLRepeatMeta {
  useIndex?: boolean
  iteratorName?: string
  dataSourceIdentifier?: string
  iteratorKey?: string
}

export interface UIDLConditionalNode {
  type: 'conditional'
  content: {
    node: UIDLNode
```

```
    reference: UIDLDynamicReference
    value?: string | number | boolean
    condition?: UIDLConditionalExpression
  }
}

export interface UIDLConditionalExpression {
  conditions: Array<{
    operation: string
    operand?: string | boolean | number
  }>
  matchingCriteria?: string
}

export interface UIDLElementNode {
  type: 'element'
  content: UIDLElement
}

export interface UIDLElement {
  elementType: string
  name?: string
  key?: string // internal usage
  selfClosing?: boolean
  ignore?: boolean
  dependency?: UIDLDependency
  style?: UIDLStyleDefinitions
  attrs?: Record<string, UIDLAttributeValue>
  events?: UIDLEventDefinitions
  children?: UIDLNode[]
}

export type UIDLNode =
  | UIDLDynamicReference
  | UIDLStaticValue
  | UIDLRawValue
  | UIDLRepeatNode
  | UIDLElementNode
  | UIDLConditionalNode
  | UIDLSlotNode

export type UIDLAttributeValue = UIDLDynamicReference | UIDLStaticValue

export type UIDLStyleValue = UIDLAttributeValue | UIDLNestedStyleDeclaration

export type UIDLStyleDefinitions = Record<string, UIDLStyleValue>

export type UIDLEventDefinitions = Record<string, UIDLEventHandlerStatement[]>

export interface UIDLEventHandlerStatement {
  type: string
  modifies?: string
```

```
  newState?: string | number | boolean
  calls?: string
  args?: Array<string | number | boolean>
}

export interface UIDLDependency {
  type: 'library' | 'package' | 'local'
  path?: string
  version?: string
  meta?: {
    namedImport?: boolean
    originalName?: string
    importJustPath?: boolean
  }
}

export interface WebManifest {
  short_name?: string
  name?: string
  icons?: Array<{ src: string; type: string; sizes: string }>
  start_url?: string
  background_color?: string
  display?: string
  orientation?: string
  scope?: string
  theme_color?: string
}

export interface Mapping {
  elements?: Record<string, UIDLElement>
  events?: Record<string, string>
  attributes?: Record<string, string>
  illegalClassNames?: string[]
  illegalPropNames?: string[]
}
```

APPENDIX D. JSON DEFINITION FOR MAPPING OPENUIDL TO HTML5

This appendix reproduces the definition of mapping rules from OpenUIDL to HTML5, expressed as a JSON (JavaScript Object Notation) file [46]. Each transformation is a one-to-one mapping from a OpenUIDL specification to a HTML5 tag. Note that this file can be edited without without altering the three disciplines of the OpenUIDL environment. For example, instead of a one-to-one mapping, a decision tree for selecting widgets could be specified depending on the logic to be pursued.

```
{
  "elements": {
    "container": {"elementType": "div"},
    "group":     {"elementType": "div"},
    "main":      {"elementType": "main"},
    "header":    {"elementType": "header"},
    "nav":       {"elementType": "nav"},
    "section":   {"elementType": "section"},
    "article":   {"elementType": "article"},
    "aside":     {"elementType": "aside"},
    "footer":    {"elementType": "footer"},
    "text":      {"elementType": "span"},
    "textblock": {"elementType": "p"},
    "strongtext":{"elementType": "strong"},
    "formattedtext": {"elementType": "pre"},
    "label":     {"elementType": "label"},
    "heading1":  {"elementType": "h1"},
    "heading2":  {"elementType": "h2"},
    "heading3":  {"elementType": "h3"},
    "heading4":  {"elementType": "h4"},
    "heading5":  {"elementType": "h5"},
    "heading6":  {"elementType": "h6"},
    "image":     {"elementType": "img",
      "attrs": {
        "src": { "type": "dynamic", "content": { "referenceType":
    "attr", "id": "url" } }
      },
      "selfClosing": true
    },
    "textinput": {
      "elementType": "input",
      "attrs": {"type": { "type": "static", "content": "text" }},
      "selfClosing": true
    },
    "passwordinput": {
      "elementType": "input",
      "attrs": {"type":  { "type":"static", "content": "password"
   }},
      "selfClosing": true
    },
    "numberinput": {
```

```
      "elementType": "input",
      "attrs": {"type": { "type": "static", "content": "number" }
   },
      "selfClosing": true
    },
    "checkbox": {
      "elementType": "input",
      "attrs": {"type": { "type": "static", "content": "checkbox"
    }},
      "selfClosing": true
    },
    "radiobutton": {
      "elementType": "input",
      "attrs": {"type": { "type": "static", "content": "radio" }}
   ,
      "selfClosing": true
    },
    "textarea": {   "elementType": "textarea"},
    "link": {
      "elementType": "a",
      "attrs": {
        "href": {
          "type": "dynamic",
          "content": { "referenceType": "attr", "id": "url" }
        },
        "rel": {
          "type": "static",
          "content": "noreferrer noopener"
        }
      }
    },
    "navlink": {
      "elementType": "a"
    },
    "button": {
      "elementType": "button"
    },
    "form": {
      "elementType": "form",
      "attrs": {
        "method": { "type": "dynamic", "content": { "
   referenceType": "attr", "id": "type" } },
        "action": { "type": "dynamic", "content": { "
   referenceType": "attr", "id": "url" } }
      }
    },
    "list": {
      "elementType": "ul",
```

```
      "children": [{
        "type": "repeat",
        "content": {
          "node": {
            "type": "element",
            "content": {
              "elementType": "li",
              "name": "item",
              "children": [{
                "type": "dynamic",
                "content": {
                  "referenceType": "local",
                  "id": "item"
                }
              }]
            }
          },
          "dataSource": { "type": "dynamic", "content": { "
    referenceType": "attr", "id": "items" } },
          "meta": {
            "useIndex": true
          }
        }
      }]
    },
    "dropdown": {
      "elementType": "select",
      "children": [{
        "type": "repeat",
        "content": {
          "node": {
            "type": "element",
            "content": {
              "elementType": "option",
              "name": "option",
              "children": [{
                "type": "dynamic",
                "content": {
                  "referenceType": "local",
                  "id": "item"
                }
              }]
            }
          },
          "dataSource": { "type": "dynamic", "content": { "
    referenceType": "attr", "id": "options" } },
          "meta": {
            "useIndex": true
```

```
          }
        }
      }]
    },
    "video": {
      "elementType": "video"
    },
    "audio": {
      "elementType": "audio"
    },
    "picture": {
      "elementType": "picture",
      "children": [
        { "type": "dynamic", "content": { "referenceType": "
   children", "id": "children" } },
        { "type": "static", "content": "This browser does not
   support the image formats given" }
      ]
    },
    "source": {
      "elementType": "source",
      "attrs": {
        "src": { "type": "dynamic", "content": { "referenceType":
    "attr", "id": "url" } }
      },
      "selfClosing": true
    },
    "svg": {
      "elementType": "svg"
    },
    "separator": {
      "elementType": "hr"
    }
  },
  "events": {},
  "attributes": {},
  "illegalClassNames": ["", "Component"],
  "illegalPropNames": ["", "this", "prop", "props", "state", "
   window", "document"]
}
```

Received February 2020; revised March 2020; accepted April 2020